UNIVERSITY OF CALIFORNIA
SANTA CRUZ

ACCESS CLASSIFICATION FOR RACE DETECTION OPTIMIZATION

A dissertation submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Dustin Rhodes

March 2019

The Dissertation of Dustin Rhodes
is approved:

Cormac Flanagan, Chair

Stephen Freund

Peter Alvaro

Lori Kletzer
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Access Classification for Race Detection Optimization

by

Dustin Rhodes

Multicore architectures are an increasingly important technique used to achieve increased performance in modern CPUs [31]. While increasing the number of cores in a chip leads to easy performance benefits when running multiple applications in parallel, writing multithreaded programs has proven to be difficult even for experienced programmers [63, 1]. This difficulty stems, in part, from the loss of fundamental abstractions available in single threaded programs. These lost abstractions include sequential consistency [91], atomicity [50], and determinism [76, 35, 64]. Research has gone into recovering these abstractions in multithreaded environments, but all are difficult to recover in the presence of data races. Therefore, eliminating data races has become a key issue in writing multithreaded code.

The desire to eliminate data races has led to the development of tools to detect and/or stop data races in multithreaded programs. Unfortunately, these *race detectors* must monitor all access to shared memory to detect data races. Shared memory is a huge portion of memory, and in many modern programming languages, such as Java and C++, it comprises all of heap memory. Likewise, checks must be added to all program locations that access shared memory. The huge range of memory and the large number of program points to be monitored leads to large overheads in race detectors.

This thesis aims to increase the speed of dynamic race detectors by restricting the set of memory and program locations they monitor. In general, we aim to classify various objects or locations as *safe*, meaning that race detection checks can be omitted on these locations/objects without reducing precision. We classify memory/program locations as *safe* by either:

- an overlapping race check,

- only being *reachable* by a single thread,
- belonging to a synchronized class,
- or being protected by a synchronized class.

These safe classifications allow for their corresponding checks to be omitted or simplified, leading to performance gains in race detection and improved programmer understanding of complex multithreaded programs.

Acknowledgments

Thanks to my advisor, Cormac Flanagan; you taught me so much about the nature of programming, mathematics, and writing. I never felt I understood proofs until working with you.

Thanks to my dissertation committee, Stephen Freund and Peter Alvaro, for the motivation. Thank you, Steve, for your help when I was working on RoadRunner code. Your advice was the perfect complement to Cormac's theoretical teaching and helped me learn much more about implementation work.

Thanks to everyone in the lab for making my work environment enjoyable: Tim Disney, Tommy Schmidt, Chris Schuster, Kongposh Sapru, and Sohum Banerjea. It was fun learning from, and with, all of you.

Finally, thanks to my family. This program has been one of the hardest things I've done and you were all ready to help me whenever I needed it. Thank you, Mom and Dad, for all the proofreading of essays and practice talks. Thank you, Celeste, Bianca, and Gabe, for all the fun times we had in Santa Cruz. Thank you, Rebecca (and Pixie and Foxy), for helping me through all the times I doubted that I could finish the degree. You had confidence in me even when my confidence in myself was lowest. Thank you all for the amazing amount of love and support I received in traversing this journey.

The text of this thesis includes reprints of the following previously published material:

- BigFoot: Static check placement for dynamic race detection by Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund published in PLDI 2017 and
- Correctness of Partial Escape Analysis for Multithreading Optimization by by Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund published in FTFJP 2017

The co-authors listed in this publication directed and supervised the research which forms the basis for the thesis.

Chapter 1

Introduction

Data race conditions are a notorious problem in multithreaded software, often resulting in erroneous outputs and violations of expected correctness properties, such as sequential consistency, atomicity, and determinism. Much prior work has focused on static [2, 22, 8, 57, 10, 37, 41, 72, 100] and dynamic [87, 99, 74, 108, 79, 30, 88, 108] data race detection.

With the introduction of multiple threads comes the question of how these threads interact. One common method of interaction is for threads to share memory (as is done in most imperative and object oriented languages). This sharing introduces the possibility for two threads to access the same memory location simultaneously; this unordered memory access is a *data race*.

Data races cause some of the invariants that programmers often rely on, such as sequential consistency [91], atomicity [50], and determinism [76, 35, 64] to break down. Without these invariants, it can be difficult for a programmer to identify bugs in their programs or reason about the possible executions. Unfortunately, data races can also be difficult for programmers to identify. Static and dynamic race detection tools aid programmers in identifying data races, but come with a severe slowdown due to the need to check every memory access.

Static analyses are able to reason about all executions of a program, but generate false alarms or miss actual data races due to their necessarily conservative approximations of program behavior. In contrast, precise dynamic analyses offer a stronger guarantee of reporting a data race *if and only if* a race occurs in the observed trace. The main limitation of precise dynamic detection is performance. The most efficient precise detectors, such as DJIT$^+$ [79] and FastTrack [47], have overheads close to an order of magnitude or more, which is too high for many applications.

This thesis aims to reduce the overhead of dynamic race detection, as well as aid programmer understanding, by classifying some memory/program locations as *safe*. In general race detectors add checks to all program locations that access heap memory. A safe location is a location at which the race detection check can be elided without changing the result of race detection. This technique can cut down dramatically on the number of checks made, thus reducing the overhead of dynamic race detectors. Classifying certain accesses as safe also leaves programmers free to focus their debugging efforts on the relatively fewer memory locations that rely on more complex synchronization.

We begin with micro classifications that aim to leverage the huge amount of overlap in the access patterns of an average program. This work is published as BigFoot: Static Check Placement for Dynamic Race Detection in PLDI 2017 [82]. BigFoot improves the speed of race detection by combining and sometimes eliding race detection checks. For example, every element of an array may be accessed by a loop. Traditionally, race detection checks each of these access separately for a race. However, BigFoot can often coalesce these checks into a single check over the whole range at the end of the loop, greatly speeding up the process of race detection. Alternatively, in many cases the same memory location is accessed multiple times with no intervening locking/unlocking operations; in this case, only one access needs to be checked as if any is involved in a race, they will all are involved. We show that this work has a large effect on the efficiency of precise dynamic race detection and can reveal some usage patterns about code (such as what memory locations are always accessed together). We also provide a proof of correctness for eliding these checks.

From there we increase our scope to look at macro classifications based on the reachability of memory. While all heap memory is shared, and in a language like C can be accessed by any thread through pointer arithmetic, portions of the heap may not be reachable by all threads in memory safe languages. Heap memory that is only reachable by a single thread is known as thread-local memory. Escape analyses are used by compilers and race detectors to determine what memory is guaranteed to be thread-local throughout its life. We extend this analysis to *partial thread-local* memory, memory which may eventually escape its creating thread but has not done so yet. This optimization has recently been suggested for compilers [96] but has not been proven correct. We provide a proof that removing checks on partial thread-local memory is safe, but eliding partial thread-local locks is unsound and can introduce race conditions. This work is published as Correctness of Partial Escape Analysis for Multithreaded Optimization in FTFJP 2017 [83].

Through this work we find that while many objects are reachable by multiple threads, most are built so that they function if only a single thread is *inside* (executing method code) at a time. We call these objects *capsules* and introduce the idea of *capsule-local* memory to denote objects that are only reachable through a single capsule. Like thread-local memory capsule-local memory is also free from data races.

We provide a syntax and semantics which we use to define a capsule-local algorithm. We implement this algorithm in RoadRunner to build a filter for capsule-local accesses and analyze a series of benchmarks in order to gain a sense of how effective optimization based on capsules can be in practice. We provide a proof that capsule-local memory can not be involved in a data race and show that a majority of program locations and accesses in the JavaGrande [59] and DaCapo [12] benchmarks only access capsule-local memory.

Multithreaded code is prone to errors due, in part, to the lack of guarantees provided by the programming environment when compared to a single threaded environment. Removing race conditions makes a mulithreaded environment more closely resemble a traditional single threaded one. Race detection can be used to either verify that a given program is race-free or identify races in a racy program so they may be removed. Unfortunately, dynamic race detection has a large overhead. We reduce the overhead of precise dynamic race detectors using location and memory classifications. In particular this thesis provides:

- a race detector based on micro classifications called BigFoot that improves on the state of the art (published in PLDI 2017),
- a proof of correctness for this race detector (published in PLDI 2017),
- a proof of correctness for removing checks on partial thread-local memory (published in FTFJP 2017),
- a proof of incorrectness for partial lock elision and provide an example where compilers performing partial lock elision introduce data races (published in FTFJP 2017),
- a proof of correctness for removing checks on capsule and capsule-local memory,
- and results that show the majority of program locations are capsule-local in popular Java benchmarks.

Chapter 2

Background

2.1 Problems Associated with Data Races

Programming concurrent code can be more difficult than programming single threaded code. The difficulties inherent in parallel code come from a number of complexities that are not present in single threaded code, for example a lack of:

- sequential consistency,
- atomicity,
- and determinism.

Prior research provides fixes for these issues but most of these fixes rely on a race free execution. Therefore, we focus not on addressing these individual problems but on the detection of data races. The following section lays out the work already done to ensure that with proper race detection these problems are also solvable.

2.1.1 Sequential Consistency

Programmers rely on their code being executed in order, and as such, languages use a *sequentially consistent memory model.* The memory model of a language defines which values are legitimate return values for a read on a memory location. A sequentially consistent memory model ensures that the observable output of a program is consistent with each memory read returning the latest value written to that memory location. This property guarantees, for instance, that the code below will always print `4` and never `3`.

```
x = 4;
print(x);
x = 3;
```

Compiler optimizations may modify or rearrange program statements but only when they can maintain sequential consistency [5]. In the above example, the compiler may see that a write to `x` of `4` occurs before a read of `x`, with no intervening writes, so optimize the print statement to `print(4)`, thereby eliminating a read at run time. This optimization, while safe in a single threaded environment, is problematic if another thread is able to write to `x` between the assignment and the print statement.

In a multithreaded environment, a sequentially consistent trace is one in which the observable output of the program is consistent with a total ordering on the operations of various threads, and each read of a location returns the most recent write to that location according to this total ordering [6]. In the following example, sequential consistency dictates that the program must print `0` if it prints anything [91]. Here the two sides of the figure represent two concurrently running threads:

```
   initially x = 0, y = 0
-----------------------------
x = 1;          | if(x == 1){
if(y == 1){     |   x = 0;
  print(x);     |   y = 1;
}               | }
```

Any sequentially consistent interleaving of these two threads will either print `0` or print nothing at all. The only sequence of actions which will print anything is the left thread assigns `x` to `1`. Then the right thread assigns `x` to `0` and `y` to `1`. Finally, the left thread prints the value of x, which must be the most recent write 0. All other sequentially consistent interleavings result in no output.

However, under the current Java memory model, it is possible (both in theory and in practice) for this program to print 1. The compiler sees that there is a write to x of 1 followed by a read of x at the print statement, with no intervening assignments to x. It then optimizes the statement print(x) into print(1). This optimization causes the program to print 1 in the interleaving described above. Note, this abnormality is only possible because of the data race on variables x and y. If the code is modified to prevent this data race (as shown below), then the program behaves as expected.

```
   initially x = 0, y = 0
---------------------------
acq(l);      | acq(l);
x = 1;       | if(x == 1){
if(y = 1){   |   x = 0;
  print(x);  |   y = 1;
}            | }
rel(l);      | rel(l);
```

While inlining reads and other compiler optimizations are guaranteed to maintain sequential consistency in single threaded programs, they only guarantee sequential consistency in multithreaded programs as long as the program is race-free. However, race-freedom is not guaranteed by the Java or C++ languages and neither provide an easy way to detect if a race has occurred. As such, while debugging programs, a programmer does not know if they are able to rely on sequential consistency. Ensuring that a program has no data races using a race detector ensures sequential consistency and allows a programmer to reason about the possible return values of a memory read while debugging.

2.1.2 Race-freedom Implies Sequential Consistency

Parallel programming languages are not able to provide a sequentially consistent memory model without sacrificing efficiency. The standard memory model for single threaded programs defines the only correct value for a read of a location to be the most recent write to that location [5]. There is one, and only one, valid value for any given read. Memory models based on this principle ensure *sequential consistency*, the observable results of the program are consistent with the source code being executed in the order specified by the program.

This memory model proves too restrictive for many compiler optimizations when in a multithreaded setting. Instead, languages like Java use a more complex memory model that only offers sequential consistency in the absence of data races [7]. Under the Java memory model, the above guarantees of a memory read returning the most recent write only apply in the absence of data races. Languages like C++ use a similar memory model where the return value of a read is undefined in the presence of data races [5]. C++ is free to use undefined values in its specification but Java, being a memory safe language, can not return an undefined value. In order to define which values are valid in the presence of races, the concurrent Java memory model is complex with numerous bugs appearing over the years [5, 81].

While Java's more complex memory model does define the valid return values for a read, even in the presence of a data race, the reads are non-deterministic (they have multiple valid return values). This non-determinism greatly increases code complexity. Both the C, C++, and Java memory models are therefore problematic to reason about as a programmer in the presence of data races.

The ambiguity introduced by non-sequentially consistent memory reads is a significant loss as a programmer must now reason about multiple possible read values. Additionally, without a race detector, a programmer may not know if their program has data races and may incorrectly reason about their code as if it has none. These types of memory models make correctly identifying data races a vital part of concurrent programming, both in order to understand what return values are valid from reads as well as for enforcing sequential consistency when it is needed. In the absence of data races, the memory models of Java and C++ behave like their single threaded counterparts. Therefore, there is a large incentive to remove data races either through language features or by using race detection [4, 40].

2.1.3 Atomicity

Our above definition of sequential consistency in multithreaded environments allows for the arbitrary interleaving of the program statements of two threads. This freedom of interleaving allows for some unexpected behavior. For example:

initially `x = 0`	
`x++;`	`x++;`

Viewed as above, it seems that `x` should result in `2` regardless of which increment operation happens first. However, `x++` is not an atomic statement in most languages. Instead it consists of a read, a modify, and a write:

initially `x = 0, r1 = 0, r2 = 0`	
`r1 = x;`	`r2 = x;`
`r1 = r1 + 1;`	`r2 = r2 + 1;`
`x = r1;`	`x = r2;`

With `x++` expanded, we can see that thread 1 can begin its increment by reading `x` as `0` and storing this value in `r1`. After this read, but before the increment, it is possible for thread 2 to also read the value of `x` as `0` and store the value in `r2`. At this point, both threads increment their temporary values from `0` to `1` and write their `1` to `x` resulting in a final value of `1`, when the correct result should be `2`, essentially losing one of the increments.

This type of error is an *atomicity* error. A portion of code that the programmer views as *atomic*, or a discrete portion of code, is interleaved with another thread in a way that causes inconsistencies. Many patterns we use have an expectation of atomicity. For example:

initially `x` is not `null`

```
if(x != null){     |
  x.f();           |  x = null;
}                  |
```

In a single threaded environment, this code will never fail due to a null pointer exception. However, in a multithreaded environment, the statement `x = null` can be interleaved between the guarding `if` and the function call reintroducing a null pointer exception. If the `if` block could be marked as atomic, then a null pointer exception would once again be impossible.

In the above example, the code has both a data race and an atomicity violation. However, atomicity errors can occur even in code without data races. Atomicity violations and data races are related but not identical issues. In multithreaded applications, critical sections can be used to enforce atomicity guarantees but only in the absence of data races [73]. A region surrounded by a lock on l can not be interleaved with other regions surrounded by the same lock. Thus, lock acquire and release statements can be used to enforce some desired atomicity properties, but only in the absence of data races. In the presence of data races, a region protected by a lock on l may be interleaved with a data race thus destroying any atomicity guarantees. In this sense, a data race is often the sign of an atomicity violation. Other tools exist that aim to check atomicity properties specifically that do not handle general race detection [50, 45, 66]

2.1.4 Non-determinism

Non-determinism in code is the cause of numerous bugs [62]. Sources of non-determinism include: C programs that only crash under certain memory configurations, programs that rely on faulty external input, and multithreaded programs [76, 35, 64]. The non-determinism in multithreaded programs is difficult to avoid, appearing even in programs without data races. A system that wishes to enforce determinism on a multithreaded program must monitor every heap access and enforce a specific ordering on them. This method of enforcing determinism causes a large overhead in software systems that use it [76]. However, in the absence of data races, enforcing determinism becomes much simpler.

The non-determinism present in multithreaded programs appears in two places: first, the order in which locks are acquired by the program and second, what return values the memory model returns for racy reads. There are orders of magnitude fewer lock operations than memory accesses and so eliminating the first type of non-determinism is significantly easier. Eliminating only the non-determinism of lock acquisition is considered *weak determinism*, while eliminating both types of non-determinism is considered *strong determinism* [76].

In the absence of data races, all memory accesses are deterministic by default. Therefore, in race free programs, all non-determinism comes from the order in which locks are acquired. For example, the following code will print either `1 2` or `2 1` depending on which thread is able to acquire `l` first.

```
acq(l);     | acq(l);
print(1);   | print(2);
rel(l);     | rel(l);
```

Enforcing determinism on lock acquires and releases is much simpler and has a lower overhead than enforcing determinism on memory accesses. While weak determinism is helpful, in order to fully realize its benefits, the target program must be race-free. If a program is weakly deterministic and has no data races, then it is also strongly deterministic. Therefore, identifying and removing data races allows programmers to use the relatively low overhead weak determinism tools to enforce strong determinism without the associated overhead. In much the same way that type and memory safe languages have eliminated many of the sources of memory based non-determinism, weak determinism tools can eliminate the sources of mulithreaded non-determinism in race free programs.

2.2 Alternative Solutions to Parallel Programming

There are a variety of techniques aimed at simplifying coding for multithreaded architectures. Attempts have been made to eliminate race conditions and their associated problems, either through hardware, language constructs, or static/dynamic detection of race conditions. While we aim to improve dynamic race detection as a solution, we outline some of the other attempts at solving the problem below.

2.2.1 Transactional Memory

For years, databases have been used to share information across multiple program instances without races. One of the key concepts that gives databases their ease of use is the *transaction* [32]. A transaction is a unit of computation that either completes or does nothing. The transaction either commits the entire unit of work to the database or none of it (in the case where it is interrupted midway), thus preventing the atomicity errors seen in Section 2.1.3.

In many modern languages, we have no way of marking a portion of code as atomic, and many of the operations that we would assume are atomic are not. For instance, as noted above, the code `x++` appears atomic but is actually a sequenced read, add, and write.

Transactional memory aims to address issues of atomicity and, more generally, of data races by applying the idea of transactions to program memory. In the case of multiple threads incrementing the same variable, one thread would *commit* its change to memory before the other thread. This commit causes the second thread to rerun its computation (rereading the value of `x`, incrementing it, and then writing the value back), thereby preventing the lost update.

There have been attempts to introduce transactional memory at both the hardware level [14] and software level [93]. However, there are difficulties with both techniques. Hardware level implementations are expensive to implement, while software level implementations are too slow. Both suffer from the difficulty of efficiently handling both small and large transactions [14]. Some attempts at specific data-structure level transactions have been made with specialized functions to support both small and large transactions [95], but a general solution to the problem does not appear imminent.

2.2.2 Restricting Compiler Optimizations

Compilers are free to move program statements as long as sequential consistency is maintained in a single threaded environment. However, in the presence of data races, even trivial code migration can break sequential consistency, as seen in Section 2.1.1. Compilers either have to forgo optimization of code or define a looser standard than sequential consistency. As we saw in Section 2.1.2, most have chosen to loosen the standard.

Some work has been done in rewriting compiler optimizations to enforce sequential consistency even in the face of data races, however, these lead to less optimization [18]. Other work has focused on writing compilers that handle parallelization themselves. These compilers are able to ensure sequential consistency while still allowing optimization. However, it is difficult for a compiler to determine what can be safely multithreaded statically, as this decision relies on reference reachability, a statically undecidable problem. In practice, large amounts of code that could be parallel are serialized in these systems [77].

The far more prevalent solution has been to use memory models that only offer sequential consistency in the absence of data races. While many current compilers use this technique, such as the C/C++ and Java compilers, defining what is allowed in the presence of data races has still proven problematic. Bugs in Java's concurrent memory model have been found and the memory model is agreed to be highly complex [81].

While loosening the restriction of sequential consistency to sequential consistency in the absence of data races solves many compiler optimization problems, it introduces a new one. As compilers are unable to detect data races, the programmer has no idea if their code behaves sequentially consistently without the added help of a race detector.

2.2.3 Type Systems

To overcome the limitations of static decidability, some languages use type systems to parallelize code without programmer input. This technique fits well in functional programming languages, such as Data Parallel Haskell [24] or Jade [84], in which memory access is already restricted by the type system. In these systems, because data races only happen through shared memory, all pure computations can safely be parallelized across multiple threads with no risk of introducing data races. However, in languages where memory reads and writes are not already restricted by the type system, new type systems must be introduced to determine what code is safe to parallelize.

For example, deterministic parallel Java [16, 15] is a Java-like programming language that introduces the idea of dividing memory into *regions* and implements new syntax to allow a programmer to mark loops or sections of code that may be run in parallel. The introduction of regions allows the type checker to verify that code marked to be run in parallel only operates on disjoint portions of memory. By forcing threads to work on disjoint regions of memory, race-freedom is guaranteed even in the absence of locking operations.

However, in order to provide such strong static guarantees, the burden placed on the programmers by the type system is relatively high. Fields must be manually labeled by region. These regions divide memory into distinct sections and parallel code must show that every thread operates on a distinct region. For algorithms that use tree structures in which memory can be divided into increasingly smaller portions, this technique works well, but it can struggle for algorithms where defining disjoint memory regions is non-trivial.

Likewise, the Rust [69] type system also ensures that threads do not operate on the same memory. In the Rust type system, references are only reachable by a single thread. When a reference is shared, it is *lent*. Lending a reference removes the original thread's copy, thereby preventing duplication of references across threads. Rust also introduces multiple traits to allow the programmer to define objects that can not be shared, can be shared safely, or can be shared with some qualifications. By using its idea of ownership, Rust is able to implement safe multithreading at the cost of a more restrictive type system.

These solutions work well for specific use cases, but a general, easy to use, type system does not exist for Java or C++. New languages can aid in this area but do not solve the concurrency problems associated with programs already written in C++ or Java as there is no easy conversion solution.

2.3 Race Detection

2.3.1 Overview

Languages which used shared memory such as C++ and Java require programs to be race-free or give up sequential consistency. Unfortunately, these languages lack a built in mechanism to identify data races in executions. As such, a programmer can never rely on any guarantees of sequential consistency, atomicity, or determinism as they have no way of knowing if a data race has occurred in a given execution or not.

Race detection is an area of research which aims to bridge this gap by adding the ability to detect data races to either the language runtime or a separate debugger. Race detectors provide a flexible approach that can be turned on during debugging and turned off again in production. During debugging, a race detector will provide either the location of a data race that the programmer can fix or a guarantee that no races occurred. If no races occurred, then standard tools and reasoning may be relied upon. Ideally, race detection would eventually be rolled into a language runtime to allow data races to act as exceptions (as suggested in Conflict Exceptions: Simplifying Concurrent

Language Semantics with Precise Hardware Exceptions for Data-Races [67]).

Static analyses are able to reason about all executions of a program, but they generate false alarms or miss actual data races due to their necessarily conservative approximations of program behavior. In contrast, precise dynamic analyses offer a stronger guarantee of reporting a race condition *if and only if* a race occurs in the observed trace. The main limitation of precise dynamic detection is performance. The most efficient precise detectors, such as DJIT^{+} [79] and FastTrack [47] have overheads close to an order of magnitude or more, which is too high for many applications. We aim to limit this drawback of precise dynamic detectors by cutting down on the large number of checks that need to be made and the large regions of memory to be monitored.

2.3.2 Race Definition

As we have seen, identifying data races is a vital part of concurrent Java or C++ programming. Intuitively, a data race occurs when two threads access the same memory location at the same time. Languages define a data race more formally by constructing a *happens-before* graph for a program trace.

A multithreaded program consists of a number of concurrently running threads. These threads can access memory, acquire or release locks, spawn off new threads, or join back into their spawning thread. A single execution of a multithreaded program can then be formalized as a trace consisting of the sequence of operations performed by the various program threads. For simplicity, we omit all operations that are not involved in race conditions. Therefore these traces are made up of memory accesses, locking operations, and the creation of new threads.

We reason about the timing of operations within a trace by using the *happens-before relation* $<_\alpha$. Two operations are ordered by happens-before if:

- the two operations are performed by the same thread,
- or the two operations acquire/release the same lock.

The happens-before relationship is defined as the smallest transitively-closed relation for which the above hold. If two actions from the same thread appear in a trace, the first always happens before the second. Likewise if a release of a lock appears in a trace before an acquire of that lock the release action happens before the acquire action.

If a happens before b, then b happens after a. If two operations in a trace are not related by the happens-before relation, then they are considered *concurrent*. Two memory access *conflict* if they both access (read or write) the same variable, and at least one of the operations is a write. Using this terminology, a trace has a data race if it has concurrent conflicting accesses.

In addition to the above definitions, we define a *synchronous-free region* (SFR) as any section of code where there are no synchronization operations. For instance, the code between a lock acquire and its later release forms an SFR. These SFR form blocks that all happen "at the same time" with regard to the operations of other threads. If an operation a in SFR A happens-before operation b of another thread, then operation a' also in SFR A also happens-before b.

2.3.3 Program Traces

Data races are a property of a particular program execution. A target program may never have a data race, have a data race on only some executions, or contain data races on every execution. A program that has a data race on some executions is still sequentially consistent on those executions where it does not have a data race. Our dynamic analyses work the same way, they report a race if one occurs in the current execution not if the program may race on some execution.

A *program trace* contains information about a specific execution. As a program executes, it performs a number of actions out of a selection of possible actions. By taking all of these actions in sequence we generate a trace. Our analyses monitor different program properties and so have different sets of actions, but all include the following:

- t accesses $o.f$ - the thread t is has accessed (read or write) the field f of object o,

- t acquires o - the thread t has acquired the lock o,
- t releases o - the thread t has released the lock o,
- t_1 forks t_2 - the thread t_1 forks off the thread t_2

While a trace has an intrinsic order, however, this order is not the same as the happens before order. For example, in the trace 1 *access* $o.f$. 2 *access* $o.f$ it appears that thread 1 accessed $o.f$ first and thread 2 accesses it second. In reality these accesses are unordered and this is a data race. In order to see when operations of different threads occur in relation to each other it is necessary to use the trace to construct a happens before graph. For an exact definition of data races, see the previous section.

Traces are also useful for defining atomicity and determinism properties. All of our analyses work on program traces and can either be thought of as either happening alongside execution or working on stored traces. In practice, we make use of the RoadRunner (section 2.4) framework which allows for implementations of our algorithms to run alongside execution.

2.3.4 Race Detection Precision

Race detectors have a variety of different correctness properties based on their intended use case. These range from heuristics based analyses that may report a race when none is present or vice versa to completely precise detectors that report a race if and only if a race occurred. To solve the problems outlined in section 2.1 the runtime environment must guarantee a lack of races; we therefore do not consider methods which produce false negatives (a race occurs but it is not reported). In theory, false positives still let us solve many of the issues related to parallelism and shared memory, but they also limit the scope of writeable programs (some correct programs won't run because the runtime believes a race has occurred when it has not) and so we try to avoid analyses which give false positives.

Even within this restricted scope, there is still latitude on the precision of race detectors. Specifically, we define precise (no false positives or false negatives) race detectors as either trace precise or address precise. An *address precise* race detector will report every address involved in a race for a given trace if and only if that address is involved in a race. A *trace precise* race detector has a weaker notion of correctness and will report a race in a given trace if and only if that trace contains a race. A trace precise race detector will also report the address of the first race, but it is possible that this first race corrupts the analysis state so future races may be missed. We never run into this corruption in practice as detailed in section 3.1 and memory safety in languages like Java can sometimes prevent it entirely.

Either of these correctness properties are sufficient for our goals, as in either case, if the race detector reports that no races occurred, then it is guaranteed that no races have occurred. For identifying races in existing programs, address precision can be helpful for solving multiple races without needing to rerun the analysis. However, in all practical tests we have run both precision types give the same results.

2.3.5 Static Race Detection

Many static analysis techniques for identifying races have been explored, including systems based on types [2, 8, 57], model checking [25, 107, 71] and dataflow analysis [41], as well as scalable whole-program analyses [72, 100]. While static race detection provides the potential to detect all race conditions over all program paths, decidability limitations imply that any static race detector that catches all races will also produce false alarms. Many of the mentioned static analyses are either unsound by design or unsound in their implementations to reduce the number of spurious warnings (see, *e.g.*, [2, 41]).

Many of the above systems make use of types in some way to aid in their static detection. However, the type burden placed on the programmer is less than the systems in Section 2.2.3. For example, static systems such as Type-Based Race Detection for Java [44] allow for fewer type annotations when compared to systems like Deterministic

Parallel Java (DPJ) [60]. However, unlike DPJ, these systems are imprecise, missing some data races and reporting false positives on others. In addition to having fewer annotations, the type system used in Type-Based Race Detection remains much closer to the standard Java type system. Annotations show which locks protect which pieces of data and parallelism is done manually, as opposed to DPJ where annotations divide memory into regions and parallelism is achieved through added syntax [60].

In addition to purely static systems, other systems aim to statically improve dynamic race detectors' performance. For example, Gross *et al.* present a global static analysis to improve the precision and performance of a LockSet-based detector [98]. Their analysis aims to statically eliminate dynamic checks on objects that it can statically guarantee are not involved in races. However, as their system relies on an imprecise dynamic race detector, their system both misses races and reports spurious warnings.

Choi *et al.* present a different global analysis for removing run-time race checks for accesses guaranteed to be race-free [29]. They also introduce a "weaker than relation", that defines some memory locations as weaker than others. The weaker memory location is accessed *if and only if* the stronger memory location has already been accessed (and checked) within the current SFR. In general, the fact that detecting data races relies heavily on aliasing information means most static detection is difficult.

2.3.6 Imprecise Dynamic Checking

In contrast to static checking, dynamic checking has access to precise aliasing information and does not suffer from decidability limitations. However, for speed reasons, many dynamic checkers remain imprecise. For example, the lockset algorithm for Eraser and Goldilocks [40, 87] serves as the basis for much of the imprecise dynamic checking work.

The lockset algorithm does not track the happens-before relation and so is not precise. Instead, it tracks a set of guarding locks for all memory location. A new memory location starts out with the set containing all locks. At a memory access, the algorithm finds the intersection of the memory location's set of locks and the set of locks currently held by the accessing thread. The algorithm then assigns the resulting set to be the new set of locks protecting that location. If the set of locks protecting a location ever drops to the empty set an error is reported.

This strategy works well for programs that have been written with lockset in mind, as, in these programs, a field not protected by a lock is very likely to be part of a data race. However, there are many situations where a lockset may be empty but a data race has not occurred. Consider the following example:

```
     initially x.f = 1
-------------------------
acq(L);     |
            | acq(L);
a = x.f;    |
            | acq(M);
rel(L);     |
            | x.f = 3;
acq(M);     |
            | rel(L);
b = x.f;    |
            | rel(M);
rel(M);     |
```

In the above code, no data race is present on `x`, as thread 2 can only write to `x.f` if it holds both locks. However, threads are free to read from `x.f` if they hold only one of the two locks. Under the lockset algorithm, the first read operation assigns the set of locks protecting `x` to `L`. The second read intersects the set containing only `L` with the set containing only `M`. This intersection causes the set of locks protecting `x` to become the empty set, and an error is reported even though no race has occurred. Another area where false positives occur in lockset algorithms is from thread-local objects which may be safely accessed with no locks held. Lockset based algorithms must either accept that all thread-local accesses will be false positives or add in heuristics to avoid these false positives which may introduce false negatives.

While lockset algorithms are relatively fast compared to precise algorithms, they report false positives in real world applications. Unfortunately, distinguishing between a false positive and a true data race has shown to be difficult [49], limiting the usability of race-detectors that report false positives. Some work has gone into automatically differentiating between these false positives and true positives [108, 75, 79], but false positives can never be completely eliminated with these methods. Choi *et al.* present a modified lockset algorithm that instead ensures that all access locksets are mutually intersecting [29]. This modification removes many false positives but is still not precise.

Other systems such as MUVI [65] use heuristics about which sets of variables are accessed together in order to compress the total number of accesses that need to be checked. This compression can either be done without loss of information resulting in the same precision as the original, or more aggressively to increase speed but lose precision. BigFoot uses some of these techniques to speed up its precise race detection but without the use of heuristics and with a vector clock based analysis in mind.

In order to solve problems with sequential consistency, atomicity, and determinism, the runtime must be able to guarantee that the current program trace does not have any races. For this reason, tools that give false negatives are unsuited for the job and tools with false positives mean that valid programs may be rejected.

2.3.7 Precise Dynamic Checking

Precise dynamic race detectors are able to detect data races with no false positives (never reporting a race when none are present) or false negatives (never missing races that have occurred during an execution). The detector reports a race *if and only if* a race occurred on the current execution. This accuracy is a large benefit given the difficult nature of identifying false positives.

The basis for precise dynamic checking stems from the *vector clock* algorithm [79]. This algorithm relies on vector clocks that keep one "time" for each thread in the target program. In practice, a vector clock is an array of integers. Each thread keeps a vector clock, recording its own time and a time for each of the other threads.

Threads, locks, and memory locations have their own vector clock. Locks' and threads' vector clocks are updated on synchronization operations that impose a happens-before order between operations of different threads. For example, when thread u releases lock m, the vector clock algorithm updates the clock of m to be the clock of u and increments u's clock. If a thread t subsequently acquires m, the algorithm updates the clock of t to be the maximum of the clocks of m and t since subsequent operations of thread t now happen after the release operation in thread u.

Finally, in order to correctly identify races, the vector clock algorithm keeps two vector clocks for each memory location, a read and a write vector clock. These vector clocks record the time of the last read and write to the that location by each thread. A read on a memory location by thread t is race-free if it happens after the last write of each other thread. A write on a memory location by thread t is race-free if it happens after the last read and write by all other threads.

Since their introduction, vector clock algorithms have been heavily optimized. FastTrack brings the running time from 20x to 8.5x (when comparing RoadRunner implementations in Java) by noting the fact that, given no data races, writes to a memory location are totally ordered and thus only the most recent access needs to be recorded and compared. This insight cuts the running time of vector clock checking from linear in the number of threads to constant time for all writes and most reads [47].

FastTrack makes this optimization by introducing the notion of *epochs*. Instead of recording an integer time for each thread, an epoch contains the thread I.D. and time of the last access. An epoch is sufficient to track happens before when events are totally ordered, because a verification that the current access comes after the previous access also means the current access comes after all accesses before it due to transitivity. In all instances except read sharing of a location, accesses must be totally ordered, so in practice this optimization leads to an asymptotic speedup.

Other efforts have focused on both reducing the amount of redundant checks [49], eliminating provably safe checks, and compressing the state of the vector clocks [103]. These optimizations reduce both the memory and runtime overhead of vector clock algorithms with state-of-the-art algorithms having slowdowns of 7.1x [103]. All the vector clock algorithms we examine or implement in this thesis are precise.

2.4 RoadRunner

Data races are possible at any shared memory access. Therefore, most dynamic algorithms aim to instrument memory accesses to do some additional form of checking when memory is accessed. To facilitate the ease of development and to allow for consistent comparisons across various algorithms, the RoadRunner framework was developed [48].

RoadRunner provides a framework for designing and testing dynamic race detection algorithms. The user defines functions for events (read, write, acquire, release, etc.) that are inserted into user code statically by the RoadRunner system. For example, RoadRunner takes code of the form: `x = y.f; y.f = z` and adds in checks, transforming it into the form:

```
readCheck(y.f); x = y.f; writeCheck(y.f); y.f = z;
```

To track the timing information of various locations, RoadRunner also adds *shadow state* to every field. A standard vector clock based race detector stores a read and a write vector clock for every field. The optimization seen in FastTrack [47] changes all write vector clocks and many read vector clocks to epochs (from an array of integers to a single integer). A lockset based algorithm usually keeps a set of the guarding locks for every field access. These shadow states make up the bulk of memory overhead for the various race detectors.

RoadRunner naively adds in checks at all memory accesses and synchronization operations. While this is sufficient for a precise algorithm, it is also overly conservative. Therefore, some systems such as RedCard [49] provide more complex check insertion algorithms that limit the number of redundant checks. These systems then work to show that their analysis remains precise even with reduced checking. The work of this thesis is mainly focused on reducing the number of checks that are inserted.

The examples used in this thesis all assume they are being run using a RoadRunner-like system. It is also possible to implement checks directly, however, this strategy makes comparing runtime overheads between algorithms difficult. Different analyses may be implemented for different languages, computing platforms, or virtual machines and may run on a standard virtual machine or a modified one [48].

2.4.1 Difficulties in Verifying Race-Freedom

Ensuring that a program is race-free is a boon in reasoning about multithreaded programs. In the absence of races the memory models of Java and C++ guarantee sequential consistency just like their single-threaded counterparts. Without data races, atomicity and determinism become enforceable with low overhead using existing tools. In fact, data races themselves have been shown to correlate with program bugs [62, 91] in real world code.

However, race-freedom is difficult to verify because every heap memory access is potentially a race and therefore must be checked. Modern attempts at eliminating race conditions come in the form of both static [2, 22, 8, 57, 10, 37, 41, 72, 100] and dynamic [87, 99, 74, 108, 79, 30, 88, 79] race detection algorithms. Static race detection reasons about multiple control flow paths but is, by necessity, imprecise. Dynamic race detectors are often sound but slow, although some trade precision for speed. Dynamic sound race detectors are precise in that they never report false positives, but are still relatively slow to run on large software, with leading algorithms such as Fasttrack having an 8.5x overhead [47]. Much of this slowdown comes from the need to perform a check on every memory access [47].

This thesis aims to increase the speed of precise dynamic race detectors through static and dynamic optimizations. Specifically, we aim to remove checks that are:

- redundant (they have already been checked or can be coalesced),
- on thread-local variables,

- or local to fully synchronized objects (such as memory accesses inside `Vector` to its array storage).

Chapter 3

BigFoot Micro Classifications

3.1 Introduction

Precise dynamic data race detectors provide strong correctness guarantees but have high overheads because they generally keep analysis state in a separate *shadow location* for each heap memory location, and they check (and potentially update) the corresponding shadow location on each heap access. The BigFoot dynamic data race detector uses a combination of static and dynamic analysis techniques to *coalesce* checks and compress shadow locations. With BigFoot, multiple accesses to an object or array often induce a single coalesced check that manipulates a single compressed shadow location, resulting in a performance improvement over FastTrack of 61%.

In this section, we present an optimized precise dynamic data race detection algorithm, BigFoot, that mitigates these overheads as follows:

1. Rather than keeping a distinct shadow location for each field in an object, or each entry in an array, BigFoot employs compressed representations using fewer shadow locations per object/array.

2. Rather than checking and updating shadow location metadata at each memory access of the target program, BigFoot uses a sophisticated static analysis to optimize *check placement* in the target code. In particular, it statically eliminates

Standard Race Checks

```
class Point {
  int x, y, z;

  void move(int dx, int dy, int dz) {
    int tmp;
    CheckRead(this.x); tmp = this.x;
    CheckWrite(this.x); this.x = tmp + dx;

    CheckRead(this.y); tmp = this.y;
    CheckWrite(this.y); this.y = tmp + dy;

    CheckRead(this.z); tmp = this.z;
    CheckWrite(this.z); this.z = tmp + dz;


  }
}

void movePts(Point[] a, int lo, int hi) {
  for(int i = lo; i < hi; i++) {
    CheckRead(a[i]);
    a[i].move(1, 1, 1);
  }
}
```

BigFoot Race Checks

```
class Point {
  int x, y, z;

  void move(int dx, int dy, int dz) {
    int tmp;
    tmp = this.x;
    this.x = tmp + dx;

    tmp = this.y;
    this.y = tmp + dy;

    tmp = this.z;
    this.z = tmp + dz;

    CheckWrite(this.x/y/z);
  }
}

void movePts(Point[] a, int lo, int hi) {
  for(int i = lo; i < hi; i++) {
    a[i].move(1, 1, 1);
  }
  CheckRead(a[lo..hi]);
}
```

Figure 3.1: Check placement for precise data race detection.

redundant checks where possible and statically combines multiple checks into a single *coalesced* check covering multiple fields or array indices.

Figure 3.1 compares BigFoot's static check placement algorithm to the standard approach of performing a check at each access. In the `move` method, a typical race detector would instrument each of the six accesses with a check verifying that the access is race-free. In contrast, BigFoot determines that the read check in each read-modify-write sequence is redundant with the check on the subsequent write, in the sense that the read will be involved in a data-race only if the write is also in a race. Thus, the read checks are not necessary to validate whether a trace is race-free.

Furthermore, BigFoot combines the three write checks into a single coalesced check CheckWrite(`this.x/y/z`) covering all three fields. Coalescing field checks in this manner is particularly helpful because it enables static shadow location compression for objects. In particular, suppose that all checks on `Point` objects are coalesced checks of the form CheckWrite(`p.x/y/z`) or CheckRead(`p.x/y/z`). BigFoot can then safely combine the shadow locations for the three fields into a single shadow location, and the coalesced checks then perform a single check-and-update operation on that shadow location, in contrast to the six checks on three shadow locations required by the traditional approach.

BigFoot optimizes array checks similarly, as shown in the method `movePts` in Figure 3.1. That code iterates over all array indices in `a` from `lo` to `hi` and moves each corresponding `Point`. In contrast to a standard dynamic race detector, which separately checks each array read, BigFoot coalesces these checks into the single check CheckRead(`a[lo..hi]`) after the loop. Here, `lo..hi` denotes the closed-open interval $\texttt{lo}, \texttt{lo} + 1, \ldots, \texttt{hi} - 2, \texttt{hi} - 1$.

To efficiently handle such coalesced checks, BigFoot again employs a compressed representation for array shadow locations. In contrast to objects however, this compressed representation is chosen and adaptively refined at run time. Specifically, an array like `a` is initially represented as a "coarse-grained" single shadow location covering all array elements. A call such as `movePts(a,0,a.length)` generates a coalesced check CheckRead(`a[0..a.length]`) covering all array elements, which]is processed at run time by checking and updating that array's single shadow location. If a subsequent call `movePts(a, 0, a.length/2)` generates a check CheckRead(`a[0..a.length/2]`) covering just half the array elements, the BigFoot run time would refine the shadow state for `a` to be two shadow locations, each covering half of `a`. That check is then handled by appropriately updating the first of these two shadow locations.

BigFoot's adaptive mechanism for arrays, modeled after SlimState [103], enables compressed array representations under a variety of common access patterns including block-based and strided accesses. If those patterns are not followed, BigFoot reverts to the "fine-grained" representation of a shadow location for each array element.

Race	Check Motion and Coalescing		Red. Check	Metadata Compression		Run-Time
Detector	objects	arrays	Elimination	objects	arrays	Overhead
FastTrack	no	no	no	no	no	7.3x
RedCard	no	no	static	static proxy	static proxy, global	6.0x
SlimState	no	dynamic	no	no	dynamic	6.0x
SlimCard	no	dynamic	static	static proxy	dynamic	5.1x
BigFoot	**static**	**static**+dynamic	static, better	static proxy	dynamic	2.5x

Figure 3.2: Comparison to prior precise dynamic race detectors, and SlimCard (which combines RedCard and SlimState).

Imprecisions in BigFoot's static analysis may lead to sub-optimal check placement, as in the following example:

```
for(int i = 0; i < a.length; i++) {
  if(predicate()) {
    a[i].move(1, 1, 1);
    CheckRead(a[i]);
  }
}
```

BigFoot's method-local analysis will not statically coalesce the array checks because it cannot statically determine which elements are accessed. At run time, suppose `a` has a single shadow location when this code runs. If `predicate()` always returns true, then all indices in `a` are accessed, and we'd like to preserve the coarse-grained representation to save both space and time. To do so, BigFoot's run time defers checks on arrays, and instead dynamically records a per-thread footprint of which indices have "pending" checks. BigFoot "commits" the footprint for a thread and checks the corresponding shadow locations for races when the thread next performs a synchronization operation. This dynamic footprinting technique allows BigFoot to keep a single shadow location for the array `a`, even in the presence of a scenario like the above that is not amenable to static coalescing.

Figure 3.2 compares BigFoot to several prior precise race detection algorithms: FastTrack, RedCard (which statically eliminates some redundant checks and compresses shadow state), SlimState (which dynamically compresses array shadow state), and SlimCard (which combines the RedCard and SlimState analyses, as described in Section 3.6). All were implemented in the RoadRunner framework for Java [48]. The key innovations of BigFoot, namely static check motion and coalescing, provide substantial performance improvements, particularly when combined with existing static and dynamic shadow compression techniques.

Detection Precision A data race detector is *trace-precise* if it correctly determines whether a given trace has a race condition or not. A trace-precise race detector is additionally *address-precise* if it can also determine all addresses that have race conditions. Using this terminology, FastTrack and SlimState are address-precise. Our BigFoot core algorithm is also address-precise, as we discuss in Section 3.3. Our BigFoot implementation, however, uses additional check placement optimizations for which one data race may prevent the detection of a subsequent race. Consequently, our implementation is trace-precise but not address-precise, as described in Section 3.5.[1] In practice, the BigFoot implementation was address-precise in all our experiments.

We also note that since BigFoot defers checking until after accesses occur, a data race may be detected only after it has happened. This introduces several subtleties related to precision. First, we currently assume for simplicity that all loops terminate and consider all unchecked exceptions to be programming errors. Thus, a race preventing a loop from terminating or causing an unchecked exception may be missed since the deferred check is never reached. However, we did not see this occur in practice, and we discuss analysis extensions to cover these items in Sections 3.3 and 3.5. In addition, if a data race can corrupt the race detector's analysis state, it may similarly go undetected [11], but for type-safe languages like Java, this cannot happen.

Contributions The primary contributions of this chapter are:

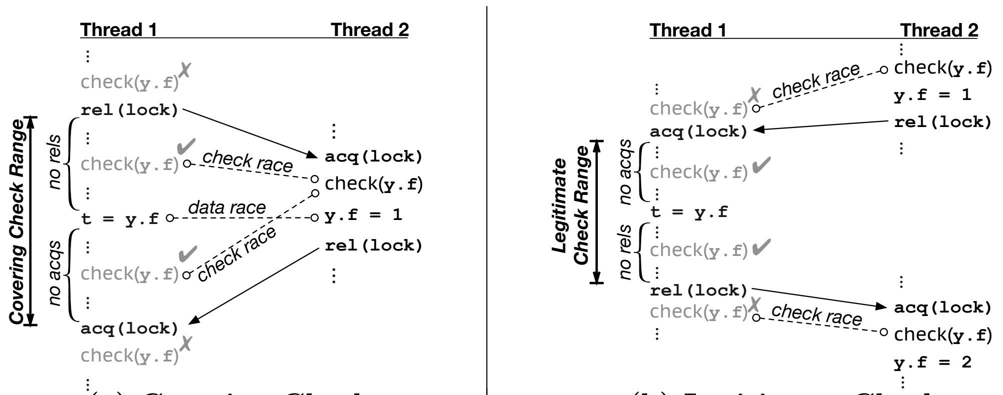
- We define a theory of precise check placement for dynamic race detection and describe a core static analysis to optimize check placement (Sections 3.2 and 3.3).
- We integrate static field proxy compression and dynamic array shadow compression techniques to further reduce run-time overhead (Section 3.4).
- We present our BigFoot prototype for Java (Section 3.5).
- We show that BigFoot's static analysis scales well (requiring on average less than 0.2s per method processed) and reduces run-time overhead from 7.3x (for FastTrack) to 2.5x, an improvement of 61% (Section 3.6).

[1]RedCard and SlimCard exhibit similar precision properties for the same reasons.

```
1: acq(lock);
2: x = b.f;
3: rel(lock);
4: y = b.f;
5: check(b.f);
6: acq(lock);
7: z = b.f;
8: rel(lock);
```

$$
\begin{array}{r}
\emptyset \bullet \{\mathtt{b.f}^{\diamond}\} \\
\{\mathtt{b.f}^{\triangleleft}\} \bullet \{\mathtt{b.f}^{\diamond}\} \\
\emptyset \bullet \{\mathtt{b.f}^{\diamond}\} \\
\{\mathtt{b.f}^{\triangleleft}\} \bullet \emptyset \\
\{\mathtt{b.f}^{\triangleleft}, \mathtt{b.f}^{\surd}\} \bullet \emptyset \\
\{\mathtt{b.f}^{\triangleleft}, \mathtt{b.f}^{\surd}\} \bullet \{\mathtt{b.f}^{\diamond}\} \\
\{\mathtt{b.f}^{\triangleleft}, \mathtt{b.f}^{\surd}\} \bullet \emptyset \\
\emptyset \bullet \emptyset
\end{array}
$$

Figure 3.3: A code fragment with precise checks, and the corresponding BigFoot analysis contexts from Section 3.3. (All variables are thread-local, and objects thread-shared.)



(a) Covering Checks

(b) Legitimate Checks

Figure 3.4: Precise and imprecise check placement locations.

3.2 Theory of Check Placement

A key design goal of the core BigFoot algorithm is that the checks inserted into a target program enable address-precise data race detection. That is, BigFoot must insert checks that are sufficient to detect all data races but that never report false alarms. Reasoning about this requirement can be subtle. For example, the code in Figure 3.3 contains a single check that enables precise data race detection for all three accesses, but it may not be immediately apparent why this is the case.

In this section, we develop a theory of check placement to characterize exactly where checks must be performed to avoid false negatives and false positives. To simplify our exposition, we initially do not distinguish between read and write accesses (although our implementation extends these ideas to do so, as described in Section 3.5).

Given an execution trace of a program, we say the trace has a *data race* if it has two accesses to the same memory location that are not ordered by the happens-before relation, which is defined in the usual fashion [61, 79].

Similarly, a trace has a *check race* if it has two checks to the same memory location that are not ordered by happens-before. A precise check placement algorithm must ensure that any execution trace of the target program has a data race *if and only if* it has a check race.

Figure 3.4(a) illustrates where checks must be performed in a trace to guarantee all data races are detected. The trace shown has a data race because the happens-before edges (shown as solid arrows) generated by synchronization operations do not order the two accesses to `y.f`, as indicated by the dashed edge. Any check performed by Thread 1 in the *Covering Check Range* will trigger a check race corresponding to that data race. However, checks outside that range will not, resulting in a false negative because the access race would have no corresponding check race.

With this intuition, we say that a check *covers* an access to the same location by the same thread if the check either:

- precedes the access with no intervening release, or
- succeeds the access with no intervening acquire.

Note that we treat acquire and release differently, as they serve as sources and sinks for synchronization edges in the happens-before graph, respectively. Returning to Figure 3.3, the single check thus covers all three accesses in any trace generated by this code. We show in the appendix that if each access in a program has a covering check, then any trace with a data race also has a check race. That is, access coverage guarantees no missed races.

Figure 3.4(b) illustrates where checks may be performed in a trace to guarantee all check races indicate data races. This trace has no data race because the three accesses to `y.f` are ordered by happens-before edges. Similarly, the checks inside the critical section of Thread 1 (marked *Legitimate Check Range*) produce no check races. However, a check outside this range produces a check race, which would be a false alarm since there is no corresponding data race.

We say that a check is *legitimate* for an access to the same location by the same thread if the check either:

- precedes the access with no intervening acquire, or
- succeeds the access with no intervening release.

For example, in Figure 3.3, the check is legitimate for the second access, but not the first or third.

With these notions of legitimacy and coverage, we say a trace has *precise checks* if each access is covered by some check (no missed races) and each check is legitimate for some access (no false alarms). A program has precise checks if all possible execution traces have precise checks.

3.3 Optimizing Check Placement

We next describe our static analysis for optimizing the placement of precise checks.

3.3.1 BFJ Language and Semantics

We formalize our ideas in terms of the idealized language BFJ (BigFoot Java) shown in Figure 3.5. A program P contains a sequence of class definitions $\overline{D}$ and a collection of concurrent threads $s_1 \| ... \| s_n$. Each class definition D contains field and method declarations. Each field declaration is simply a field name f. Each method declaration $m(\overline{x})\{s;\ \texttt{return}\ z\}$ includes a unique method m, formal parameters $\overline{x}$, and a body s followed by a return of the local variable z. We omit static types and local

$$
\begin{array}{llll}
P & \in & \mathit{Program} & ::= \ \overline{D}\ s_1 \| \ldots \| s_n \\
D & \in & \mathit{Defn} & ::= \ \texttt{class}\ c\ \{\ \overline{f}\ \overline{\mathit{meth}}\ \} \\
\mathit{meth} & \in & \mathit{Method} & ::= \ m(\overline{x})\ \{\ s;\ \texttt{return}\ z\ \}
\end{array}
$$

$$
\begin{array}{llll}
s & \in & \mathit{Stmt} ::= & \texttt{skip} \ \mid\ s; s \ \mid\ \texttt{if}\ \mathit{be}\ s\ s \\
 & & \mid & \texttt{loop}\{\ s;\ \{\ \texttt{if}\ \mathit{be}\ \texttt{break}\ \};\ s\ \} \\
 & & \mid & x = e \ \mid\ x \leftarrow y \ \mid\ \texttt{acq}(y) \ \mid\ \texttt{rel}(y) \\
 & & \mid & x = \texttt{new}\ c \ \mid\ y.f = x \ \mid\ x = y.f \\
 & & \mid & x = \texttt{new\_array}\ z \ \mid\ y[z] = x \ \mid\ x = y[z] \\
 & & \mid & x = y.m(\overline{z}) \ \mid\ \texttt{check}(C)
\end{array}
$$

$$
\begin{array}{llll}
e & \in & \mathit{Expr} & ::= \ x \mid v \mid e = e \mid \ldots \\
\mathit{be} & \in & \mathit{BoolExpr} & \subseteq \ \mathit{Expr} \\
C & \in & \mathit{PathSet} & ::= \ 2^{\mathit{Path}} \\
p & \in & \mathit{Path} & ::= \ x.f \mid x[r] \\
r & \in & \mathit{StridedRange} & ::= \ e..e\,{:}\,e
\end{array}
$$

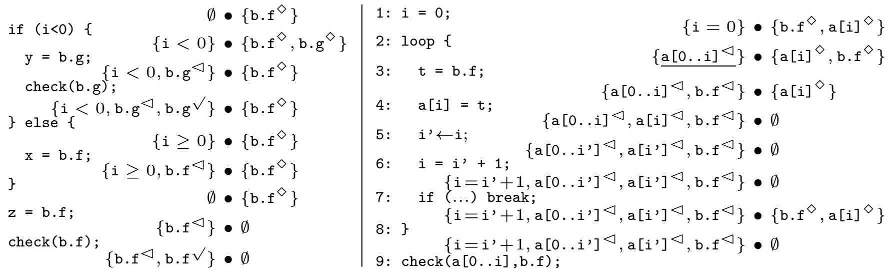
$$
\begin{array}{rcl rcl}
c & \in & \mathit{ClassName} & f & \in & \mathit{FieldName} \\
m & \in & \mathit{MethodName} & x, y, z & \in & \mathit{Var}
\end{array}
$$

Figure 3.5: BFJ Syntax.

variable declarations, which are orthogonal to our formal development. We leave the set of expressions e unspecified but assume it includes at least `null`, boolean values, and local variables.

To facilitate our technical development, BFJ statements are in A-normal form [52] and include a loop construct with the exit test in the middle of the loop body. We motivate and describe the renaming operator $x \leftarrow y$ below.

BFJ includes the statement `check(`C`)` to explicitly check for races on each heap location described by a path $p \in C$. A path of the form $x.f$ describes an object field, and path of the form $x[r]$ describes array accesses, where r is a *strided range* of the form "$b..e\,{:}\,k$" represents the set of indices $\{b + ik \ :\ b \leq b + ik < e\}$ to be checked. We use b and $b..e$ to abbreviate singleton ("$b..(b+1)\,{:}\,1$") and continuous ("$b..e : 1$") strided ranges, respectively. We defer distinguishing read checks and write checks until Section 3.5.



```
                          ∅ • {b.f◇}
if (i<0) {
                    {i < 0} • {b.f◇, b.g◇}
  y = b.g;
              {i < 0, b.g◁} • {b.f◇}
  check(b.g);
        {i < 0, b.g◁, b.g✓} • {b.f◇}
} else {
                    {i ≥ 0} • {b.f◇}
  x = b.f;
              {i ≥ 0, b.f◁} • {b.f◇}
}
                          ∅ • {b.f◇}
z = b.f;
                     {b.f◁} • ∅
check(b.f);
               {b.f◁, b.f✓} • ∅
```

```
1: i = 0;
                                               {i = 0} • {b.f◇, a[i]◇}
2: loop {
                                            {a[0..i]◁} • {a[i]◇, b.f◇}
3:   t = b.f;
                                      {a[0..i]◁, b.f◁} • {a[i]◇}
4:   a[i] = t;
                             {a[0..i]◁, a[i]◁, b.f◁} • ∅
5:   i'←i;
                           {a[0..i']◁, a[i']◁, b.f◁} • ∅
6:   i = i' + 1;
                {i=i'+1, a[0..i']◁, a[i']◁, b.f◁} • ∅
7:   if (...) break;
                {i=i'+1, a[0..i']◁, a[i']◁, b.f◁} • {b.f◇, a[i]◇}
8: }
                {i=i'+1, a[0..i']◁, a[i']◁, b.f◁} • ∅
9: check(a[0..i],b.f);
```

Figure 3.6: Analysis contexts and check placements for BFJ method bodies containing (a) an `if` statement and (b) a loop.

Analysis Contexts

The BigFoot analysis is intraprocedural, analyzing and inserting checks into each method one at a time. Within each method, the analysis infers a *context* $H \bullet A$ for each program point that describes the known *history properties* H and *anticipated properties* A at that point:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{Context} ::= H \bullet A \qquad H \subseteq \mathit{History} \quad A \subseteq \mathit{Anticipated}} \\
h \in \mathit{History} & ::= & \mathit{be} \mid p^{\triangleleft} \mid p^{\checkmark} \\
a \in \mathit{Anticipated} & ::= & p^{\diamond}
\end{array}
$$

These properties capture the following notions:

- Boolean expressions *be* from, *e.g.*, branch tests.
- Past accesses $p^{\triangleleft}$, meaning that path p was previously accessed, with no subsequent release. The analysis must ensure there is a corresponding covering check.
- Past checks $p^{\checkmark}$, meaning that p was previously checked within the method, with no subsequent release.
- Anticipated accesses $p^{\diamond}$, meaning that the continuation after the program point will access p (and therefore check p), with no intervening acquire.

3.3.2 Check Placement Algorithm Overview

The BigFoot check placement algorithm defers checks as long as possible and only inserts them into the program code when they cannot be further deferred without risking false alarms or missed data races; thus checks are only placed before synchronization operations and control flow merge points, and at the ends of methods and threads.

To illustrate how BigFoot uses context information to place checks, we examine the analysis contexts in Figure 3.3. As in all BFJ code, the variables in this snippet are local and cannot be changed by other threads, although they may point to shared objects.

BigFoot adds a past access $p^{\lhd}$ to the history whenever the code accesses p, and before an acquire it inserts a check for any past access $p^{\lhd}$ with no past covering check $p^{\surd}$, as at line 5. Since the acquire signifies the end of that past access's covering check range, placing the check any later would introduce the potential for missed data races.

At each release, BigFoot removes each past access $p^{\lhd}$ from the history. The release signifies the end of the legitimate check range for those accesses, and placing checks for them any later would introduce the potential for false alarms. "Forgetting" a past access $p^{\lhd}$ like this typically requires BigFoot to place a covering check before the release, but there are two situations when no check is needed: (1) a covering check has already occurred ($p^{\surd}$ is in the history), as at line 8; or (2) we anticipate a later access to the same location, as at line 3. The anticipated later access (and hence its covering check) will occur before leaving the original access's covering check range at the next acquire. Each check $p^{\surd}$ must also be forgotten at a release because that check does not cover any subsequent access to p.

Anticipated access information flows backwards, and anticipated accesses in an acquire's post-history must be removed from its pre-history because checks covering those future accesses will not cover accesses prior to the acquire.

We now examine the if statement in Figure 3.6(a). The merged context $\emptyset \bullet \{\texttt{b.f}^{\diamond}\}$ after the if describes properties holding after both branches, and it omits past accesses occurring only on one branch. BigFoot must ensure a covering check exists for any such "forgotten" past access. That necessitates checking b.g in the "then" branch, after which it is permissible to simultaneously forget both the past access and past check on b.g when leaving the if. In contrast, bz.f is anticipated at the end of the "else" branch, and we skip checking it at that point because the later access will have a check covering both accesses.

Figure 3.6(b) illustrates how loops are handled. To simplify our analysis, we require that the target x of any assignment be a "fresh" variable not mentioned in the preceding history, as the assignment would otherwise invalidate that history information. The operation i' $\leftarrow$ i copies the value of i into a fresh variable i' and replaces all mentions of i in the history by i', thereby ensuring i is afterwards fresh, that is, not mentioned in the history. BigFoot inserts renaming statements on demand, but for simplicity our presentation assumes any necessary renamings already exist.

BigFoot places all necessary checks at line 9 after the loop using the following technique. First, BigFoot synthesizes a loop invariant history that captures the set of accesses that have been performed whenever execution reaches line 2. The invariant for our example is the underlined history $H_{inv} = \{\underline{\texttt{a[0..i]}^{\triangleleft}}\}$. On entry to the loop, H_{inv} holds because i $= 0$, meaning no array elements have been accessed. On the loop back edge, H_{inv} is entailed by the loop body's final history $\{\texttt{i}=\texttt{i'}+1, \texttt{a[0..i']}^{\triangleleft}, \texttt{a[i']}^{\triangleleft}, \texttt{b.f}^{\triangleleft}\}$.

BigFoot defers checks until after the loop whenever possible. In this case, the history at the loop exit on line 7 contains $\texttt{a[0..i']}^{\triangleleft}$ (the invariant rewritten due to the renaming of i to i' at line 5) and $\texttt{a[i']}^{\triangleleft}$ (the similarly rewritten access from line 4). That history context captures all accesses that must be checked after the loop. Given that i' = i + 1, BigFoot places the single check of a[0..i] at line 9 to cover all array accesses from inside the loop.

BigFoot requires no global analysis to move the checks out of the loop because all variables referenced in the code are local and cannot be changed by other methods or threads.

This example also demonstrates that anticipation is crucial for moving some checks out of loops. At the end of the loop on line 8, the history contains $\texttt{b.f}^{\triangleleft}$, but the back edge returns to loop head on line 2, where $\texttt{b.f}^{\triangleleft}$ is not in the history. This would normally necessitate placing a check on `b.f` inside the loop before the back edge. However, since $\texttt{b.f}^{\diamond}$ is anticipated at the loop head, we can avoid checking `b.f` inside the loop and defer the check until after the loop.

Checks deferred until after a loop may never be executed if the loop diverges. We currently assume all loops terminate but could alternatively include a termination analysis and treat potentially non-terminating loops specially by, for example, periodically committing deferred checks inside the loop.

3.3.3 Check Placement Rules

We formalize BigFoot's check placement algorithm as the judgment $\vdash s : H{\bullet}A \to H'{\bullet}A'$ defined in Figure 3.7. The contexts $H{\bullet}A$ and $H'{\bullet}A'$ are the pre- and post-contexts of s. The analysis is a combined forward/backward analysis; history properties flow forward from *pre-history* H to *post-history* H', while anticipated properties flow backwards from *post-anticipated* A' to *pre-anticipated* A.

For conciseness, we do not express check placement as a rewriting transformation on program syntax. Instead, we assume that a pre-transformation has already inserted a check $\texttt{check(}C\texttt{)}$ wherever one may be required. The goal of the check placement algorithm is then to resolve each *path set variable* C into the appropriate set of paths to be checked at that point. The rules for $\vdash s : H{\bullet}A \to H'{\bullet}A'$ include antecedents constraining each C appropriately.

Context Entailment and Ordering Our rules use the notation $h \in H$ for the usual syntactic notion of set membership for history properties. In addition, we introduce

$\boxed{\vdash s : H \bullet A \to H' \bullet A'}$

(We assume $x \notin \mathit{Vars}(H)$ in the rules modifying x)

$$
\begin{array}{llcllll}
[\textsc{Skip}] & \vdash \texttt{skip} & : & H \bullet A & \to & H \bullet A & \\
[\textsc{Acq}] & \vdash \texttt{check}(C); \texttt{acq}(x) & : & H \bullet \emptyset & \to & (H \cup C^{\checkmark}) \bullet A & \text{where } C = \mathit{Checks}(H, \emptyset) \\
[\textsc{Rel}] & \vdash \texttt{check}(C); \texttt{rel}(x) & : & H \bullet A & \to & (H \setminus \{\_^{\checkmark}, \_^{\triangleleft}\}) \bullet A & \text{where } C = \mathit{Checks}(H, A) \\
[\textsc{Assign}] & \vdash x = e & : & H \bullet A[x := e] & \to & (H \cup \{x = e\}) \bullet A & \text{where } x \notin \mathit{Vars}(e) \\
[\textsc{Rename}] & \vdash x \leftarrow y & : & H \bullet A[x := y] & \to & H[y := x] \bullet A & \\
[\textsc{New}] & \vdash x = \texttt{new } c & : & H \bullet (A \setminus x) & \to & H \bullet A & \\
[\textsc{A-New}] & \vdash x = \texttt{new\_array } z & : & H \bullet (A \setminus x) & \to & H \bullet A & \\
[\textsc{Write}] & \vdash y.f = x & : & H \bullet (A \cup \{y.f^{\diamond}\}) & \to & (H \cup \{y.f^{\triangleleft}\}) \bullet A & \\
[\textsc{A-Write}] & \vdash y[z] = x & : & H \bullet (A \cup \{y[z]^{\diamond}\}) & \to & (H \cup \{y[z]^{\triangleleft}\}) \bullet A & \\
[\textsc{Read}] & \vdash x = y.f & : & H \bullet (A \setminus x \cup \{y.f^{\diamond}\}) & \to & (H \cup \{y.f^{\triangleleft}\}) \bullet A & \\
[\textsc{A-Read}] & \vdash x = y[z] & : & H \bullet (A \setminus x \cup \{y[z]^{\diamond}\}) & \to & (H \cup \{y[z]^{\triangleleft}\}) \bullet A &
\end{array}
$$

[If]

$$
\frac{\begin{array}{cc}
H_1 = H_{in} \cup \{be\} & \vdash s_1 : H_1 \bullet A_1 \to H_1' \bullet A_{out} \\
H_2 = H_{in} \cup \{\neg be\} & \vdash s_2 : H_2 \bullet A_2 \to H_2' \bullet A_{out} \\
C_1 = \mathit{Checks}(H_1', H_1' \sqcap H_2', A_{out}) & C_2 = \mathit{Checks}(H_2', H_1' \sqcap H_2', A_{out}) \\
A_{in} = H_1 \bullet A_1 \sqcap H_2 \bullet A_2 & H_{out} = (H_1' \cup C_1^{\checkmark}) \sqcap (H_2' \cup C_2^{\checkmark})
\end{array}}{\vdash \texttt{if } be\ \{s_1;\ \texttt{check}(C_1)\}\ \{s_2;\ \texttt{check}(C_2)\} : H_{in} \bullet A_{in} \to H_{out} \bullet A_{out}}
$$

[Seq]

$$
\frac{\begin{array}{c}
\vdash s_1 : H_1 \bullet A_1 \to H_2 \bullet A_2 \\
\vdash s_2 : H_2 \bullet A_2 \to H_3 \bullet A_3
\end{array}}{\vdash s_1; s_2 : H_1 \bullet A_1 \to H_3 \bullet A_3}
$$

[Loop]

$$
\frac{\begin{array}{cc}
\multicolumn{2}{c}{\vdash s : H_{inv} \bullet A_{in} \to H \bullet A_{inv}} \\
H_{back} = H \cup \{\neg be\} & H_{out} = H \cup \{be\} \\
C_{in} = \mathit{Checks}(H_{in}, H_{inv}, A_{in}) & H_{in} \cup C_{in}^{\checkmark} \sqsupseteq H_{inv} \\
C_{back} = \mathit{Checks}(H_{back}, H_{inv}, A_{in}) & H_{back} \cup C_{back}^{\checkmark} \sqsupseteq H_{inv} \\
H_{back} \vdash A_{inv} \sqsubseteq A_{in} & H_{out} \vdash A_{inv} \sqsubseteq A_{out}
\end{array}}{\begin{array}{c}\vdash \texttt{check}(C_{in}); \texttt{loop\{}\ s;\ \texttt{\{ if } be \texttt{ break \}};\ \texttt{check}(C_{back})\ \texttt{\}} : \\ H_{in} \bullet A_{in} \to H_{out} \bullet A_{out}\end{array}}
$$

[Call]

$$
\frac{\begin{array}{c}
C = \mathit{Checks}(H, H \setminus \mathit{KillSetHistory}(m), A) \\
H' = (H \cup C^{\checkmark}) \setminus \mathit{KillSetHistory}(m) \\
A = A' \setminus x \setminus \mathit{KillSetAnticipated}(m)
\end{array}}{\vdash \texttt{check}(C);\ x = y.m(\overline{z}) : H \bullet A \to H' \bullet A'}
$$

[Stmt]

$$
\frac{\begin{array}{c}
\vdash s : \emptyset \bullet A \to H \bullet \emptyset \\
C = \mathit{Checks}(H, \emptyset)
\end{array}}{\vdash s; \texttt{check}(C)}
$$

[Method]

$$
\frac{\vdash s}{\vdash m(\overline{x})\ \{\ s; \texttt{return } z\ \}}
$$

[Class]

$$
\frac{\forall\ meth \in \overline{meth}.\ \vdash meth}{\vdash \texttt{class } c\ \{\ \overline{f}\ \overline{meth}\ \}}
$$

[Program]

$$
\frac{\begin{array}{c}
\forall D \in \overline{D}.\ \vdash D \\
\forall i.\ \vdash s_i
\end{array}}{\vdash \overline{D}\ s_1 \| \ldots \| s_n}
$$

Figure 3.7: Check Placement Rules.

a richer notion of *history entailment* ($H \vdash h$) that accounts for other information in H. For example, if $H = \{z[i]^{\triangleleft}, i = j\}$ then we can safely infer that H entails $z[j]^{\triangleleft}$, written $H \vdash z[j]^{\triangleleft}$. Similarly, we introduce *anticipated entailment* ($H \bullet A \vdash a$), as in $\{i < 10\} \bullet \{x[0..10]^{\diamond}\} \vdash x[0..i]^{\diamond}$. Our implementation uses Z3 [33] to reason about entailment.

While history and anticipated sets could be ordered by the subset relation ($\subseteq$), we employ a stronger ordering ($\sqsubseteq$) based on entailment to achieve greater precision:

$$H_1 \sqsubseteq H_2 \quad \textit{iff} \quad \forall h \in H_1.\ H_2 \vdash h$$

$$H \vdash A_1 \sqsubseteq A_2 \quad \textit{iff} \quad \forall a \in A_1.\ H \bullet A_2 \vdash a$$

These orderings generate corresponding meet operators, where the meet on anticipated sets additionally takes history sets to reason about entailment.

$$\begin{aligned} H_1 \sqcap H_2 \quad &= \{h \in H_1 \cup H_2 : H_1 \vdash a, H_2 \vdash a\} \\ H_1 \bullet A_1 \sqcap H_2 \bullet A_2 &= \{a \in A_1 \cup A_2 : H_1 \bullet A_1 \vdash a, H_2 \bullet A_2 \vdash a\} \end{aligned}$$

Analysis Rules The analysis rules are somewhat complex due to their bidirectional nature and the subtle properties being captured. We present the technical details of our core rules below, but subsequent sections do not assume an in depth understanding of all of their details.

[**Rel**]**:** Since past accesses need to be checked before a release, this rule targets the syntax
`check(`C`);rel(`x`)` and uses the function

$$\mathit{Checks}(H, A) = \{\ p\ :\ p^{\triangleleft} \in H, H \not\vdash p^{\surd}, H \bullet A \not\vdash p^{\diamond}\ \}$$

to ensure that the path set C contains any path p that was accessed ($p^{\triangleleft} \in H$) but not yet checked and is not anticipated. (If p is anticipated, then the future check on the anticipated access serves as the covering check for the past access.)

The post-history removes (1) all prior checks (denoted $\_^{\surd}$) because these checks do not cover accesses after the release and (2) all prior accesses (denoted $\_^{\triangleleft}$) because we are leaving the legitimate check range for them.

[**Acq**]**:** This rule for `check(`C`);acq(`x`)` ensures C contains any path p that was accessed but not checked. The post-history contains the newly checked paths (where $C^{\surd}$ abbreviates $\{p^{\surd} \mid p \in C\}$). The pre-anticipated set must be empty because any anticipated access would need to occur before this acquire.

[**Read**]**:** This rule matches the syntax $x = y.f$. To simplify our analysis, we require that the target of any assignment be to a "fresh" variable not mentioned in the pre-history H, as the assignment would otherwise invalidate that history information. The

[Read] rule adds past access $y.f^{\lhd}$ to the post-history. The pre-anticipated paths become $A \setminus x \cup \{y.f^{\diamond}\}$, where $A \setminus x$ removes all properties mentioning x from A.

[Rename]: As mentioned above, assignments can only target "fresh" variables not in H, but in some cases, *e.g.* before a loop back edge, we may need to modify an existing non-fresh variable y. We cannot simply remove y from the history, as that might remove past accesses with pending checks, such as $y.f^{\lhd}$. Instead, the renaming operation $x \leftarrow y$ copies the value of y into a fresh variable x, and replaces all mentions of y in the history H by x, with the result that y is now "fresh" (not mentioned in the history) and can be an assignment target. To illustrate this rule, consider the renaming `i` $\leftarrow$ `i'` on line 5 in Figure 3.6(b). The history prior to the renaming contains `a[0..i]`$^{\lhd}$ and `a[i]`$^{\lhd}$. After renaming, we have `a[0..i']`$^{\lhd}$ and `a[i']`$^{\lhd}$, enabling us to continue deferring the checks for those accesses.

[Write]: This rule for $y.f = x$ adds the access $y.f^{\lhd}$ to the post-history, and $y.f^{\diamond}$ to the pre-anticipated set.

[Assign]: This rule for the assignment x `=` e adds the boolean expression $x = e$ to the post-history. We require $x \notin \mathit{Vars}(e)$ to ensure the post-history does not refer to the pre-value of x. The pre-anticipated set is computed from the A via the substitution $A[x := e]$, which replaces all occurrences of x with e in each $p^{\diamond} \in A$. Since anticipated paths are not closed under this substitution, we remove from the result any syntactically ill-formed anticipated paths.

[If]: Conditionals may require checks to be placed at the end of each branch, and so this rule targets the syntax `if` *be* $\{s_1;$`check(`C_1`)`$\}$ $\{s_2;$`check(`C_2`)`$\}$. This rule first computes the post-histories H_1' and H_2' and pre-anticipated sets A_1 and A_2 for s_1 and s_2. The merged history $H_1' \sqcap H_2'$ describes properties holding after both branches but may leave out accesses that occurred only on one branch. We introduce the following variant of the *Checks* function to compute the unanticipated unchecked past accesses in H that must be checked when H is approximated by H':

$$\mathit{Checks}(H, H', A) = \{\ p : p^{\lhd} \in H, H' \not\vdash p^{\lhd}, H \not\vdash p^{\checkmark}, H \bullet A \not\vdash p^{\diamond}\ \}$$

Thus, $C_1 = \mathit{Checks}(H'_1, H'_1 \sqcap H'_2, A_{out})$ are those paths that must be checked at the end of the "then" branch, and similarly for C_2 on the "else" branch. The contexts at the end of the branches are then $H_1 \cup C_1^{\checkmark}$ and $H_2 \cup C_2^{\checkmark}$, and these are merged via $\sqcap$ to yield the final history H_{out}. The anticipated pre-context A_{in} is computed by merging together the anticipated contexts preceding s_1 and s_2.

[Loop]: Loops similarly require checks on the two paths meeting at the loop head, and this rule targets the form:

$$\texttt{check(}C_{in}\texttt{);loop\{}\ s;\ \{\ \texttt{if}\ \mathit{be}\ \texttt{break}\ \};\ \texttt{check(}C_{back}\texttt{)}\ \}$$

In this rule, H_{in} and H_{back} are the pre-histories of `check(`C_{in}`)` and `check(`C_{back}`)`, respectively, and H_{inv} is the loop-invariant history at the loop head. As in [If], the *Checks* function uses these sets and A_{in}, the anticipated set at the loop head, to compute C_{in} and C_{back}. The side conditions $H_{in} \cup C_{in}^{\checkmark} \sqsupseteq H_{inv}$ and $H_{back} \cup C_{back}^{\checkmark} \sqsupseteq H_{inv}$ ensure that properties in H_{inv} are true on all paths into the loop head.

Note that H_{inv}, H, and H_{back} are defined via mutual recursion; they are computed as part of a greatest fixed point computation over a method body. The computation is seeded with an initial conjecture for H_{inv} that is then refined via a form of predicate abstraction. (See Section 3.5.) An analogous anticipated set A_{inv} characterizing what is anticipated prior to the loop exit test is used in the computation of A_{in}.

[Call]: A method call may require checks prior to the call if the callee performs synchronization (either directly or indirectly via a nested method call). Thus we match syntax of the form `check(`C`);` $x = y.m(\overline{z})$. The function $\mathit{KillSetHistory}(m)$ denotes the set of history properties killed by the side effects of method m, and contains:

$$\begin{array}{ll} \{\ \_^{\lhd}\} & \text{if } m \text{ acquires a lock} \\ \{\ \_^{\lhd}, \_^{\surd}\ \} & \text{if } m \text{ releases a lock} \end{array}$$

The function *KillSetAnticipated*(m) describes anticipated accesses killed by m. It is $\{\_^{\diamond}\}$ if m acquires a lock and $\emptyset$ otherwise. Our implementation pre-computes *KillSetHistory*(m) and *KillSetAnticipated*(m) using a separate whole program analysis. Checks are added before the call for any unchecked accesses C that are killed by the call, and the post-history H' is derived from the pre-history H and C by removing all such killed properties.

Correctness Sketch We first formalize an operational semantics for BFJ that evaluates program $P = \overline{D}\ s_1 \| \dots \| s_n$ via a sequence of states $\Sigma_0 \rightarrow^{a_1} \Sigma_1 \rightarrow^{a_2} \dots \rightarrow^{a_n} \Sigma_n$, where Σ_0 is an initial state for P and Σ_n is a final terminating state. This evaluation sequence yields a trace $\alpha = a_1.a_2 \dots a_n$ describing the memory accesses, race checks, and synchronization operations performed by P.

We also define a judgement $\overline{D}; \alpha \Vdash \Sigma$ describing when a run-time state has correct checks in the context of an execution history α. This judgement most notably ensures that, for each thread t, the context $H \bullet A$ for thread t's current program point is consistent with Σ and α. This judgement entails the following: 1) Each expression *be* $\in H$ is true when evaluated by t in the current state Σ. 2) If $p^{\lhd} \in H$ and p denotes a memory l, there is an access to l in by t α with no later release. (Each $p^{\surd} \in H$ must have similar check). 3) Each check by t in α is legitimate for a preceding access. 4) Each access to a location l by t in α is either covered by a check, or t is still in that access's covering check range and there is some path p denoting l such that either $p^{\lhd}$ is in H or $p^{\diamond}$ is in A.

The first two requirements show that the history context soundly approximates program behavior. The third and fourth guarantee that each check performed by t is legitimate and that each access by t has either been covered by a check or will be covered by deferred check performed later in the trace.

Provided $\vdash P$, the initial state satisfies the criteria for well-formed states (*i.e.*, $\overline{D}; \epsilon \Vdash \Sigma_0$), and we show via a preservation argument that it holds for each subsequent state, including the last, *i.e.*, $\overline{D}; \alpha \Vdash \Sigma_n$. Since each thread in Σ_n has terminated and will perform no subsequent checks or accesses, the rules for $(\Vdash)$ imply that α has precise checks. Consequently, the checks in P are address-precise. That is, if $\vdash P$ and P generates a trace α, then for any address l, α has a data race on l if and only if it has a check race on l.

3.4 Check Coalescing & Shadow Compression

Post-Analysis Path Coalescing In preparation for our shadow compression algorithms, we perform one last coalescing step on each set of checks added to the program. Specifically, for each `check(`C`)` statement, we divide the paths in C into equivalence classes based on the path designator: that is, $d_1.f_1$ and $d_2.f_2$ are in the same class if d_1 and d_2 refer to the same object in the check's pre-history written $H \vdash d_1 = d_2$), and similarly for array paths.

We then coalesce each group $d_1.f_1, d_2.f_2, \ldots, d_n.f_n$ sharing equivalent designators to the *coalesced field path* $d_1.f_1/f_2/\cdots/f_n$. We also coalesce each group of paths

$$d_1[b_1..e_1 : k_1], \ldots, d_n[b_n..e_n : k_n]$$

to one array path $d_1[b..e : k]$ such that the strided range "$b..e : k$" captures the exact same set of indices as the n original strided ranges. This step necessitates solving a collection of integer constraints over program expressions, but those constraints have a form that cannot be handled by, *e.g.*, Omega [80] or effectively solved directly via Z3. Thus, to find a suitable b, e, and k, our implementation tries various combinations of the bounds and step sizes from the original strided ranges. This combinatorial approach can be expensive if there are a large number of strided ranges, but we have found it effective in practice. If a coalesced path cannot be found, we simply keep the original set of paths. We could alternatively try to divide the set into two or more coalescible subsets, but this provided little benefit in practice.

Shadow Compression A precise dynamic race detector typically maintains a distinct shadow location for each object field or array element. Thus, an object `pt` with three fields requires three shadow locations and `check(pt.x/y/z)` performs three shadow-location operations. Similarly, an array a of n elements requires n shadow locations, and `check(a[0..`n`])` performs n shadow-location operations.

However, check coalescing enables us to identify groups of shadow locations that can be compressed into a single shadow location at run time with no loss in precision. Moreover, a coalesced check covering a compressible group only requires a single shadow-location operation, yielding substantial performance benefits. Compressible locations can be identified statically or dynamically. We have found the combination of static compression for object fields and dynamic compression for array elements yields the best performance.

Static Field Compression We identify fields of a class that are compressible via a static *shadow proxy* analysis [49]. Given a class with fields `x` and `y`, field `x` is a proxy for `y` if every check `check(p.`$\cdots$`/y/`$\cdots$`)` also checks `p.x`. In this situation, any trace exhibiting a race on `p.y` will also have a race on `p.x`. Hence, we can compress the shadow locations for `x` and `y` into a single location while still being able to distinguish race-free executions from those with races.[2] Identifying field proxies requires a single pass over all checks.

[2]While this optimization guarantees that we precisely identify race-free traces, we may not identify all memory locations with races since a race on `x` may or may not imply a race on `y`. This subtlety goes away if we consider only symmetric proxy relations, *e.g.*when `y` is also a proxy for `x`.

Dynamic Array Compression We could express similar proxy relationships for array elements. For example, `a[0]` could be a proxy for all array entries `a[0..`n`]` if all checks on the array all have the form `check(a[0..`n`])`. Similarly `a[i%2]` could be the proxy for each `a[i]` if all checks have the form `check(a[0..`n`:2])` or `check(a[1..`n`:2])`. REDCARD [49] used this approach, but its static array proxy analysis failed to scale and was too imprecise to capture many proxy relationships, as we demonstrate in Section 3.6.

BIGFOOT instead makes array shadow compression choices dynamically using an extension of the approach introduced in the SLIMSTATE checker [103]. Specifically, BIGFOOT augments static array check coalescing with a complementary dynamic coalescing technique based on array footprints. For each array `a`, the BIGFOOT run time maintains a per-thread footprint of which indices must be checked prior to that thread's next synchronization operation. When a thread t performs `check(a[b..e:k])`, BIGFOOT adds the strided range `b..e:k` to t's footprint for `a`. In this way, many individual check operations that were not coalesced statically may be coalesced dynamically into a single, large footprint. At thread t's next synchronization point, its footprint for `a` is "committed" and the necessary shadow-location operations are performed to verify race freedom.

BIGFOOT initially compresses the shadow state for the entire array into a single shadow location. It then adaptively refines that representation whenever it must commit a footprint that is not consistent with the array's current representation. As in SLIMSTATE, BIGFOOT supports compression modes matching common patterns of array accesses, including block-based and stride-based patterns. SLIMSTATE processes every individual array access at run time to build its dynamic footprints. By statically coalescing checks, BIGFOOT eliminates much of that overhead.

3.5 Implementation

We have implemented our analysis in the BigFoot checker for Java. BigFoot consists of a static component (StaticBF) and a dynamic component (DynamicBF). StaticBF reads in a bytecode program and a list of classes and methods to transform, and it outputs a version of the program with explicit race checks for all object and array accesses in the specified methods. DynamicBF is the complementary dynamic race detector that reads in the instrumented program, runs it, and reports any races observed.

Extending the BFJ analysis to the full Java language is straightforward, and we describe the most important aspects of StaticBF below. BigFoot handles all basic synchronization operations present in Java, including locks, volatile variables, fork/join, and wait/notify, as described in [47].

Alias Expressions and Precision StaticBF augments BFJ's set of boolean expression *be* with heap alias expressions of the form $x = y.f$ and $x = y[z]$, which enable us to reason about aliasing when deciding entailment. Those expressions are added to the history on field/array reads and are retained as long as they are valid under the assumption that the target is race free. If an alias expression is invalidated by a data race at run time, we may miss reporting some subsequent data races (because race checks were not placed in the necessary positions), but we will always detect the initial race.

For example, consider the code fragment to the right, which includes the alias expressions recorded by StaticBF. Those alias expressions enable StaticBF to conclude `x` = `y` at the `check` operation, meaning that the check on `x.g` covers the access to `y.g`. Thus, no check on `y.g` is inserted. However, those alias assumptions could be violated by a racy write to `a.f` in between the two reads, and thus the race on `a.f` could effectively hide a race on `y.g`.

```
acq(lock);
x = a.f; // x = a.f
s = x.g;
y = a.f; // y = a.f
t = y.g;
check (a.f, x.g);
rel(lock);
```

While utlizing local alias expressions enables StaticBF to better optimize check placement, it means that, in theory, BigFoot is trace precise but not address precise. In practice, however, BigFoot was address-precise for all of our benchmark runs, which we verified via an additional dynamic analysis that checks that each observed execution trace performs precise checks (in the sense of Section 3.2).

3.5.1 StaticBF

StaticBF is built on top of the WALA analysis framework [101]. WALA represents methods as CFGs over SSA instructions and analyzes all methods in a call graph constructed using a 0-CFA analysis. To ensure method CFGs are amenable to our analysis, StaticBF performs an initial pass over the target to (1) rewrite each loop as an `if` statement containing a `do-while` loop matching BFJ's syntax, and (2) eliminate all critical edges from the CFGs (see, *e.g.*, [9]). We use Soot [97] for this pass. We also precompute *KillSetHistory* and *KillSetAnticipated* via a simple interprocedural dataflow analysis. StaticBF then inserts checks into each method using a method-local dataflow analysis.

The initial context for each program point is $\{h : h \in \mathit{History}\} \bullet \{a : a \in \mathit{Anticipated}\}$, and the analysis computes the greatest fixed point solution for those contexts according to the rules in Figure 3.7. To simplify the implementation, we compute context properties via separate passes for (1) boolean and alias expressions, (2) past accesses, (3) anticipated accesses, and finally (4) past checks and the set C for each `check(`C`)`. All passes are forward analyses, except for the anticipated accesses pass.

StaticBF handles SSA ϕ-functions as they were handled in RedCard [49]. Also as in RedCard, StaticBF tracks extended paths containing multiple field/array references (as in `a[i].f` or `b.f.g`), which are necessary for maintaining precision when merging contexts encoding equivalent aliasing facts via different local variables. We implement the entailment relations via the Z3 SMT Solver [33].

After applying the final coalescing step and static field proxy analysis described in Section 3.4, StaticBF generates a new version of the target code with the necessary checks inserted. These checks take the form of method calls into the DynamicBF run time. Paths in check statements refer to SSA variables and variables introduced via the [Rename] rule, and not the stack slots and locals present in the original bytecode. Thus, StaticBF inserts additional locals and load/store instructions to reify them in the instrumented target. Our relatively naive algorithm may introduce extraneous memory loads/stores, and we apply the Soot optimizer in a post-transformation pass to eliminate them.

Distinguishing Reads and Writes Up to this point, we have not distinguished reads and writes. However, StaticBF must do so because precise dynamic race detectors treat them differently. In particular, two concurrent accesses are considered conflicting only when at least one is a write.

To account for this, we extend our notions of legitimate and covering checks. A write check is only legitimate for a write access, but a read check is legitimate for both write and read accesses. A write check can cover write or read accesses, but a read check can only cover read accesses. In addition, contexts record whether each $p^{\triangleleft}$ and $p^{\diamond}$ is a read or write access, and whether each $p^{\surd}$ is a read or write check. The analysis rules and coalescing operations are also extended appropriately.

Loop Invariants StaticBF infers the loop invariant H_{inv} for rule [Loop] via a form of Cartesian predicate abstraction [56, 51]. Specifically, StaticBF identifies the loop's linear induction variables and trip count [104, 54] and then builds an initial set $H_{heuristic}$ of boolean constraints and past accesses consistent with that information. Since this algorithm does not reason precisely about synchronization, function calls, and various other bytecode features, it may produce some incorrect properties. Thus, StaticBF repeatedly analyzes the loop body to infer the maximal $H_{inv} \subseteq H_{heuristic}$ that is valid loop invariant as part of its dataflow analysis passes. StaticBF similarly infers the anticipated invariant A_{inv} by constructing an initial $A_{heuristic}$ and computing the maximal valid $A_{inv} \subseteq A_{heuristic}$. If no induction variables can be identified, then $A_{heuristic}$ is the empty set, and no loop invariant are inferred. Irreducible loops and complex computations may be problematic for our algorithm, but it is quite effective in practice.

Static Fields In the JVM, a thread's first access to a static field may synchronize with the declaring class's static initializer to ensure proper behavior [68]. StaticBF provides a command line flag to treat static field accesses as potential synchronization so that checks will not be deferred across them. We use this flag for several benchmarks where this matters. (Other instructions that may synchronize with static initializers, *e.g.* type casts, are handled similarly.)

Exceptions StaticBF reasons about control paths for checked exceptions [55], but assumes unchecked exceptions, such as `NullPointerExceptions`, are errors in the target program and guarantees precision only for error-free traces. This is an artifact of our current implementation and not a fundamental limitation. Unchecked exceptions could be fully handled via a more sophisticated code translation scheme inside StaticBF, but given the complexity of the resulting code, a better approach would be to integrate parts of the analysis into the JVM's exception mechanism. Our current treatment of exceptions did not lead to missed race checks in any of our benchmark experiments.

3.5.2 DynamicBF

We built our complementary DynamicBF dynamic analysis in the RoadRunner framework [48]. Dynamic footprinting and array shadow compression are implemented as in the earlier SlimState checker and we use FastTrack's adaptive epoch representation [47] for shadow locations. BigFoot follows RoadRunner's standard treatment of libraries: fields of Java's core library classes are not checked for races, and synchronization operations internal to those libraries are assumed not to be

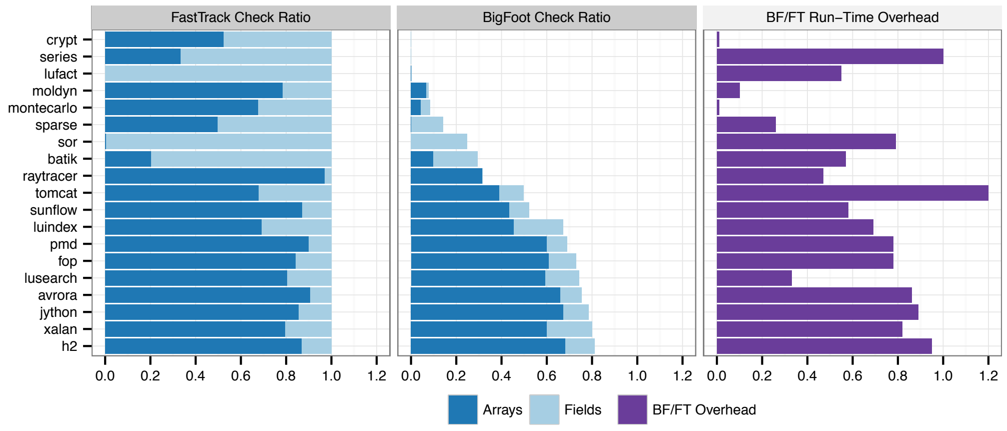


Figure 3.8: Check Ratio for FastTrack and BigFoot, and BigFoot's overhead relative to the FastTrack overhead.

used to protect any of the target's data and are ignored. However, several key library methods from `java.lang.Object` and `java.lang.Thread`, such as `Object.notify` and `Thread.start`, are treated specially as synchronizing operations. These assumptions are shared by all checkers we evaluate, and also included in StaticBF. Their violation may impact precision.

3.6 Validation

We validate BigFoot's performance by comparing it against FastTrack [47], SlimState [103], RedCard [49], and SlimCard (Section 3.6.2) on the JavaGrande [59] and DaCapo [12] benchmark suites. To facilitate comparison the detectors share as much common implementation as possible.

We configured the JavaGrande programs to use their largest data sizes and 16 worker threads. We also fixed racy barrier implementations in several of them. We configured the DaCapo benchmarks to use their default sizes, but we exclude tradebeans and eclipse because of incompatibilities with our underlying framework and other known issues [103]. We additionally exclude several specific methods from the other programs that RoadRunner cannot properly instrument because the resulting code would exceed a JVM limit on method size. Several DaCapo programs use reflection heavily. To facilitate building the call graph for those programs in StaticBF and RedCard, we used a modified version of Tamiflex [17] to eliminate reflection.

Since RoadRunner does not support the specialized class loading features used by the DaCapo test harness, we implemented a simplified version of that harness. It runs a target's workload several times in a warm up phase and then measures the running time for 10 iterations of the workload. We used that harness for the JavaGrande programs as well. We report the means of ten such trials.

We verified all race detection tools examined reported the same races (modulo variations due scheduling) manually. All experiments were performed on a 2.4GHz 16-core AMD Opteron processor with 64GB running Ubuntu Linux and Oracle's Java HotSpot 64-bit Server VM version 1.8.

3.6.1 StaticBF

BigFoot took 0.16 seconds per method on average to process the benchmark programs, as shown in Table 3.1. With careful caching of SMT solver results, only about 10% of this time was spent solving Z3 queries. Together, call graph construction for computing method kill sets and reasoning about heap and boolean constraints accounted for more than half of the running time in most cases. We have focused on implementation simplicity and high precision. More careful tuning would likely lead to significant improvements.

3.6.2 DynamicBF Time Overhead

Figure 3.8 shows, for each program, how many race checks on shadow locations FastTrack (left graph) and BigFoot (middle graph) perform relative to the number of heap accesses. FastTrack performs a check on each access, meaning its *check ratio* ($\frac{\#\ Checks}{\#\ Accesses}$) is always 1. For BigFoot, the average check ratio is 0.43, and much smaller for some programs, particularly those in which traversals over large arrays are covered by a single coalesced check. BigFoot's check ratio is also substantially lower than that of RedCard (0.73), SlimState (1.0), and SlimCard (0.76).

Table 3.1 shows the base running time for each program and the overhead of each checker. Overhead is the additional time beyond the base time necessary to check a program:

$$\text{CheckerOverhead} = \text{CheckerTime} - \text{BaseTime}$$

Comparison to FastTrack BigFoot is significantly faster than the other detectors. As shown in the last column of Table 3.1, BigFoot incurs only 39% of the overhead of FastTrack. The right-most graph in Figure 3.8 shows this improvement visually. BigFoot is most effective on programs exhibiting highly-structured access patterns to large data sets, and thus low check ratios, such as `crypt`, `moldyn`, `montecarlo`, and `sunflow`. Moving checks out of loops and coalescing them accounts for much of this improvement. BigFoot is also effective on programs with many redundant checks that can be eliminated altogether, such as `sparse`.

It is interesting to note that several programs do not follow the expected trend. For `series`, the FastTrack overhead of only 1% is mostly due to internal RoadRunner bookkeeping, which leaves little opportunity for improvement. The `lufact` benchmark performs a triangular array computation whose array accesses are readily coalesced by BigFoot, resulting in a small check ratio. However, that triangular pattern is not amenable to our online array state compression algorithm, meaning that the array's shadow representation becomes fine-grained and each coalesced check induces many shadow location operations.

In other benchmarks, such as `h2` and `avrora`, bookkeeping for synchronization operations accounts for a greater fraction of checking overhead, diminishing the benefit of optimizing memory operations with BigFoot. The degraded performance for `tomcat` appears to be caused by higher contention on interal RoadRunner data structures when using BigFoot.

Field compression via proxies accounted for about 5% of the savings in general, but over 50% of the savings in `raytracer` and `sunflow`.

	StaticBF		Dynamic Analyses										
					Time Overhead (x Base Time)					Time Overhead vs. FT			
Program	Methods Optimized (count)	Time / Method (sec)	BigFoot Check Ratio	Base Time (sec)	FT	RC	SS	SC	BF	$\left(\frac{RC}{FT}\right)$	$\left(\frac{SS}{FT}\right)$	$\left(\frac{SC}{FT}\right)$	$\left(\frac{BF}{FT}\right)$
crypt	148	0.67	0.00000028	0.39	96.21	62.41	16.87	16.11	0.07	(0.65)	(0.18)	(0.17)	(0.01)
series	144	0.10	0.000042	119.39	0.01	0.01	0.01	0.01	0.01	(1.00)	(1.00)	(1.00)	(1.00)
lufact	168	0.15	0.0022	0.68	71.67	74.31	70.53	74.08	39.53	(1.04)	(0.98)	(1.03)	(0.55)
moldyn	172	0.27	0.077	4.67	27.56	8.73	27.18	6.58	2.72	(0.32)	(0.99)	(0.24)	(0.10)
montecarlo	480	0.05	0.085	2.23	7.38	6.81	2.73	2.02	0.08	(0.92)	(0.37)	(0.27)	(0.01)
sparse	140	0.20	0.14	1.27	26.86	22.57	30.78	27.20	6.68	(0.84)	(1.15)	(1.01)	(0.25)
sor	136	0.24	0.25	0.84	13.37	13.03	15.39	13.85	10.73	(0.97)	(1.15)	(1.04)	(0.80)
batik	20,140	0.16	0.29	1.27	3.96	3.92†	4.07	4.06	2.26	(0.99)	(1.03)	(1.03)	(0.57)
raytracer	308	0.07	0.32	1.84	13.46	6.46	12.64	7.72	6.37	(0.48)	(0.94)	(0.57)	(0.47)
tomcat	27,940	0.12	0.50	0.81	2.05	1.49†	2.22	1.56	2.43	(0.73)	(1.08)	(0.76)	(1.19)
sunflow	3,088	0.21	0.52	1.44	25.94	17.13	26.12	20.50	15.14	(0.66)	(1.01)	(0.79)	(0.58)
luindex	4,728	0.07	0.67	0.54	16.35	15.75	19.00	17.64	11.34	(0.96)	(1.16)	(1.08)	(0.69)
pmd	18,604	0.18	0.69	0.93	3.08	2.98†	2.75	2.65	2.38	(0.97)	(0.89)	(0.86)	(0.77)
fop	24,756	0.15	0.73	0.44	6.51	5.12†	5.65	5.54	5.01	(0.79)	(0.87)	(0.85)	(0.77)
lusearch	3,544	0.07	0.74	0.65	19.45	22.79	7.79	7.24	6.57	(1.17)	(0.40)	(0.37)	(0.34)
avrora	9,936	0.04	0.75	7.82	1.45	1.34†	1.46	1.38	1.24	(0.92)	(1.01)	(0.95)	(0.86)
jython	81,140	0.11	0.78	4.97	9.31	9.32†	8.77	8.58	8.28	(1.0)	(0.94)	(0.92)	(0.89)
xalan	13,420	0.05	0.80	0.86	5.68	5.63†	5.62	5.43	4.64	(0.99)	(0.99)	(0.96)	(0.82)
h2	16,748	0.08	0.81	22.60	3.23	3.08†	3.20	3.23	3.07	(0.95)	(0.99)	(1.00)	(0.95)
Mean		0.16	0.43		7.26	6.00	6.03	5.05	2.47	(0.83)	(0.83)	(0.70)	(0.39)

Table 3.1: Checker performance. Mean StaticBF time and Check Ratios are arithmetic means. Mean checker overheads for FastTrack (FT), RedCard (RC), SlimState (SS), SlimCard (SC), and BigFoot (BF) are geometric means. The † symbol indicates that RedCard's proxy analysis failed to terminate within 4 hours. We turned off that analysis in those cases.

Comparison to RedCard RedCard eliminates one form of redundant check [49], namely checks on accesses where the current thread has already accessed (and checked) that location within the same release-free span. The BigFoot check placement algorithm is able to eliminate other forms of redundancy by both reasoning about anticipated accesses and moving checks. For example, BigFoot can eliminate more redundant checks and move checks out of loops, as shown in Figure 3.6.

RedCard also performs static proxy analysis, but the array component crucially depends upon globally-computed allocation-site points-to information. As such, RedCard's static analysis fails to terminate within four hours on many benchmarks, as indicated by the † symbol in Table 3.1. We use RedCard's redundancy analysis without proxies for those programs. Moreover, imprecisions in the proxy analysis limit its effectiveness even on small programs.

Overall, the check ratio and overhead reduction for RedCard were 0.73 and 17%, respectively. In contrast, the check ratio and overhead reduction for BigFoot were were 0.43 and 61%. BigFoot's ability to move checks out of loops is key to achieving

Program	Base Mem (MB)	Space Overhead				
		$\frac{\text{FT}}{\text{Base}}$	$\left(\frac{\text{RC}}{\text{FT}}\right)$	$\left(\frac{\text{SS}}{\text{FT}}\right)$	$\left(\frac{\text{SC}}{\text{FT}}\right)$	$\left(\frac{\text{BF}}{\text{FT}}\right)$
crypt	193.76	26.27	(0.97)	(0.04)	(0.04)	(0.04)
series	22.01	4.45	(1.02)	(0.58)	(0.59)	(0.57)
lufact	32.15	10.16	(1.00)	(1.10)	(1.10)	(1.11)
moldyn	16.20	5.44	(0.82)	(0.91)	(0.80)	(0.82)
montecarlo	622.83	3.67	(1.00)	(0.30)	(0.30)	(0.30)
sparse	98.11	5.64	(1.01)	(1.44)	(1.05)	(0.79)
sor	32.12	5.11	(1.00)	(1.40)	(1.40)	(2.48)
batik	44.74	3.78	(0.99)	(0.75)	(0.95)	(1.00)
raytracer	16.42	3.67	(0.96)	(0.60)	(0.57)	(0.60)
tomcat	19.59	4.81	(0.99)	(0.98)	(0.99)	(1.14)
sunflow	10.42	9.50	(0.91)	(0.93)	(0.88)	(0.86)
luindex	6.15	16.3	(0.98)	(0.96)	(0.96)	(0.52)
pmd	30.24	6.02	(1.05)	(1.02)	(1.03)	(1.09)
fop	28.07	6.35	(1.00)	(0.98)	(0.97)	(0.99)
lusearch	12.04	7.00	(1.00)	(0.57)	(0.57)	(0.57)
avrora	2.09	15.22	(1.01)	(1.01)	(1.01)	(1.01)
jython	24.06	5.97	(1.03)	(0.96)	(0.96)	(1.02)
xalan	8.20	11.00	(1.00)	(0.84)	(0.84)	(0.82)
h2	259.71	3.90	(1.06)	(1.10)	(1.10)	(0.93)
Geo Mean		6.84	(0.99)	(0.73)	(0.74)	(0.72)

Table 3.2: Checker space overhead relative to FastTrack.

this improvement, particularly when coupled with dynamic array shadow compression.

Comparison to SlimState SlimState introduced the dynamic array compression scheme we use in BigFoot, but its check ratio is 1 because it processes every access at run time. BigFoot offers two crucial improvements: 1) BigFoot eliminates many redundant checks. 2) While SlimState must process every individual array access at run time to build its footprints, BigFoot statically coalesces array checks where possible, thereby reducing the amount of run-time footprint processing and eliminating much of SlimState's dynamic footprint construction overhead. BigFoot's overhead is less than half of SlimState's as a result. Field compression, and moving field checks out of loops, contributes to the performance savings as well.

Comparison to SlimCard SlimCard combines RedCard's static check elimination and field proxy analysis with SlimState's dynamic array state compression. We did not include static proxy analysis for arrays in SlimCard because integrating the run-time bookkeeping necessary to support static array proxies [49] into SlimState's analysis led to worse performance. As a result, SlimCard has an overall check ratio of 76%, which is a few percent higher than RedCard's ratio (73%).

As expected, the combined analysis improves upon SlimState by eliminating many redundant checks and incurs only 70% of FastTrack's overhead. However, SlimCard still experiences the same overheads related to the construction of footprints at run time as SlimState. Moreover, it cannot move checks out of loops and coalesce them, which are crucial for achieving BigFoot's much better performance. SlimCard's memory overhead did not differ significantly from SlimState's or BigFoot's.

3.6.3 DynamicBF Memory Overhead

While we have focused primarily on running time, we also report the target program's memory requirements, as well as the overheads for each checker in Table 3.2. Following the methodology of earlier work [103], we measure memory as the smallest heap permitting successful execution of the target program, which we find by iteratively shrinking the JVM's maximum heap until the program crashes or fails to terminate within thrice the time to run with a 64 GB heap.

BigFoot, SlimState, and SlimCard reduce space overhead by about 26–28% when compared to FastTrack. These three tools utilize the same dynamic array compression scheme. SlimCard and BigFoot additionally uses field compression, but while field compression improved time, it did not lead to sizable space reductions. Inspection of the programs for which field compression made the greatest speed difference revealed that there were never sufficiently many objects with compressed fields alive at the same time to sizably impact overall space needs.

The limited impact of static compression on space can also be seen by comparing the space overhead of RedCard to FastTrack. The only fundamental space difference is due to RedCard's use of compression for field and array proxies, but again, there is little overall impact.

3.7 Other Related Work

In addition to RedCard and SlimState, described earlier, much work has focused on improving the performance of dynamic race detection. Many precise tools, such as DJIT$^+$ [79], use vector clocks [70], which are expensive. FastTrack introduced *epochs* [49] to reduce these overheads. A common approach for further reducing overhead is to use a single shadow location for whole arrays and objects [99, 75, 26, 79, 47, 20], although this may generate false alarms, motivating additional technology to see if a reported warning reflects a real race [23, 41].

Another approach for reducing overheads is to use sampling [19, 42, 39], again with some loss of soundness. Eraser verifies race-freedom for data that is thread-local, read-shared, or lock protected [87], and has been extended to produce fewer false alarms [75, 41, 23, 90, 106].

Several dynamic checkers defer the processing of accesses. RecPlay [85] records all locations accessed within each synchronization-free region and then verifies that concurrent regions access disjoint locations during replay. DRD [36] and ThreadSanitizer [89] similarly buffer accesses but do not infer patterns or compress shadow state. Similar buffering is also common in transactional memory systems [93]. Other work [94] uses a single shadow location for contiguous memory locations accessed within the same critical sections. However, only the first two critical sections accessing a location are considered, resulting in potential false alarms if later accesses are not correlated.

Many static analyses for identifying races have also been explored, including type-based systems [2, 8, 57], model checking [25, 107, 71] and dataflow analyses [41], as well as whole-program analyses [72, 100]. Many of the mentioned static analyses are unsound by design or unsound in their implementations to reduce the number of spurious warnings (see, *e.g.*, [2, 41]). Their focus on identifying race-free accesses rather than redundant checks also lead to different design choices in terms of precision and scalability.

Gross *et al.* present a global static analysis to improve the precision and performance of a LockSet-based detector [98]. It is primarily designed to identify objects on which no races can occur and requires global aliasing information, as well as a static approximation of the happens-before graph for the whole program. Moreover, their reliance on an imprecise race detector leads their system to both miss races and report spurious warnings. They also do not support arrays. Choi *et al.* present a different global analysis for removing run-time race checks for accesses guaranteed to be race-free [29]. Their analysis eliminates some redundant checks via a simple intra-procedural forward analysis.

Properties related to accesses or checks within release-free spans have been used in other settings. For example, the IFRit race detector uses similar insights in its notion of interference-free regions [39], which were originally designed to facilitate compiler optimizations for race-free programs [38]. The IFRit race detector monitors execution and reports a data race when multiple concurrently executing interference-free regions access the same variable. IFRit prioritizes performance over precision, and so may possibly miss races (but nicely guarantees no false alarms). IFRit uses a static analysis to insert and minimize monitor start/stop calls, which is analogous to BigFoot's check insertion algorithm. BigFoot's approach necessitates a more complex static analysis to ensure sufficient precision to perform check motion, and so is at a different point in the design space.

3.8 Summary

BigFoot leverages our theory of precise check placement to substantially improve the efficiency of dynamic data race detection. This work may enable more widespread use of data race detectors, and it opens the door for further studies on statically optimizing dynamic concurrency analyses.

One interesting direction is to extend our techniques to compress memory locations across multiple arrays or objects, which could yield further time and space savings. Another important avenue for future work is to improve StaticBF's performance by adapting it to be modular or incremental and by tailoring its data structures and decision procedures to the most common cases encountered in practice.

3.9 Proof of Correctness

We next prove that the check placement algorithm is correct. In particular it inserts checks so that any generated trace has precise checks, and so has a data race *if and only if* there is a check race detected by DynamicBF.

- Section 3.9.1 formalizes the operational semantics of BFJ.
- Section 3.9.2 shows that a trace with precise checks has a data race if and only if it has a check race.
- Section 3.9.3 formalizes a GoodChecks judgment that satisfies preservation.
- Section 3.9.4 shows that the CheckPlacement algorithm inserts checks that satisfy
  GoodChecks.
- Section 3.9.5 shows that the programs satisfying the GoodChecks judgment generate traces with precise checks.
- Section 3.9.6 shows that correctness of BigFoot.

3.9.1 Semantics

We specify the operational semantics of BFJ in Figure 3.9. This semantics evaluates a program by stepping through a sequence of states. Each state Σ consists of two components: a heap S and a collection of threads T. The heap maps locations to values, where each location $\rho.f$ or $\rho[i]$ combines an address ρ with a field name f or array index i. The heap also maps each object address ρ to the thread identifier (or *Tid*) of the thread holding the object's lock (or $\perp$ if it is not held). The thread set T maps each thread identifier $t \in \text{Tid}$ to a thread state $\langle \sigma, s \rangle$ that combines a statement s with a (thread-local) store σ mapping variables in s to values.

In the context of a set of definitions $\overline{D}$, the relation

$$\overline{D} \vdash S \cdot \langle \sigma, s \rangle \longrightarrow^{a} S' \cdot \langle \sigma', s' \rangle$$

models the effect of a single step by thread $\langle \sigma, s \rangle$ on the heap S and the thread's local state. The *Action* a captures the heap operation performed by the step. For example, if thread t accesses location $\rho.f$, a would be $t{:}\texttt{acc(}\rho.f\texttt{)}$. The special action $t{:}\epsilon$ indicates that a step has no heap effect.

Figure 3.9 defines the evaluation rules for each statement. In these rules, the heap $S[\rho.f := v]$ is identical to S except that it maps the location $\rho.f$ to the value v. Similar update operations are used on the other state components. For example, $S[\rho := t]$ updates S to indicate that the lock for the object at location ρ is held by t. The term $\sigma(e)$ evaluates an expression e using local store σ for the values of variables.

The rule [E-ChkSet] unrolls a check on a set of paths to separate checks on each path. The rule [E-ChkIndex] checks a strided range of array indices by explicitly checking the first index and generating a new check for the remainder of the strided range. Rule [E-ChkEmpty] handles empty strided array indices.

To invoke a method x = $y.m(\overline{z})$, we first look up the method m in the program definitions. We then construct a substitution θ that maps 1) m's local variables, which are the free variables of s, other than the return result variable r, to fresh names, 2) the parameters $\overline{z'}$ to the arguments $\overline{z}$, 3) the self-reference `this` to y, and 4) the return variable r to x. If s is the method body of m, $\theta(s)$ may be inserted into the evaluation context surrounding the call without variable capture. Moreover, the result of the call is placed in x, as expected.

The relation $\overline{D} \vdash \Sigma \rightarrow^a \Sigma'$ describes a single step of multithreaded program execution. That rule selects an arbitrary thread t to take a step and updates the global state Σ accordingly. As above, a captures the memory or synchronization operation performed by the step. We use the notation $t : \_$ to represent an arbitrary action by thread t.

The relation $\overline{D} \vdash \Sigma \longrightarrow^\alpha \Sigma$ denotes the reflexive-transitive closure of $\longrightarrow^a$, where the *trace* α is a sequence of actions $a_1.a_2 \ldots a_n$. Given this definition, $\overline{D} \vdash \Sigma \rightarrow^a \Sigma'$ models the arbitrary interleaving of the various threads of a multithreaded program $\overline{D}$.

For a program $\overline{D}\ s_1 \| \ldots \| s_n$, its *initial state* is $\Sigma_0 = S_0 \cdot T_0$, where

- S_0 maps all locations to `null` and all addresses to $\perp$; and
- T_0 maps each thread $t \in 1..n$ to $\langle \sigma, s_t \rangle$, where σ assigns a distinct global address to each free variable in $s_{1..n}$. Thus, free variables in $s_{1..n}$ implicitly denote potentially thread-shared objects.

$$
\begin{array}{llll}
\Sigma & \in & \textit{State} & ::= & S \cdot T \\
S & \in & \textit{Store} & = & (\text{Location} \rightarrow \text{Value}) \cup (\text{Address} \rightarrow \text{Tid}_{\perp}) \\
\rho & \in & \textit{Address} & & \\
l & \in & \textit{Location} & ::= & \rho.f \mid \rho[i] \\
u, t & \in & \textit{Tid} & ::= & 1 \mid 2 \mid \ldots \\
T & \in & \textit{Threads} & = & \text{Tid} \rightarrow \langle \sigma, s \rangle \\
\sigma & \in & \textit{Store} & = & \text{Var} \rightarrow \text{Value} \\
v & \in & \textit{Value} & ::= & \rho \mid \texttt{true} \mid \texttt{false} \mid \texttt{null} \mid i \mid \ldots \\
i & \in & \textit{Nat} & &
\end{array}
$$

$$
\begin{array}{lllll}
\alpha, \beta & \in & \textit{Trace} & ::= & \overline{a} \\
a, b, c, d & \in & \textit{Action} & ::= & t : \texttt{acc}(l) \mid t : \texttt{check}(l) \mid t : \texttt{acq}(\rho) \mid t : \texttt{rel}(\rho) \mid t : \epsilon
\end{array}
$$

$$\boxed{\overline{D} \vdash \Sigma \rightarrow^{a} \Sigma'}$$

$$\overline{D} \vdash S \cdot T[t := \langle \sigma, s \rangle] \quad \rightarrow^{a} S \cdot T[t := \langle \sigma', s' \rangle] \qquad \text{if } \overline{D} \vdash S \cdot \langle \sigma, s \rangle \rightarrow^{a} S' \cdot \langle \sigma', s' \rangle \text{ and } a = t{:}\_$$

$$\boxed{\overline{D} \vdash S \cdot \langle \sigma, s \rangle \longrightarrow^{a} S' \cdot \langle \sigma', s' \rangle}$$

$$
\begin{array}{lllll}
\overline{D} \vdash S \cdot \langle \sigma, \texttt{check}(\{x.f\}) \rangle & \longrightarrow^{t\,:\,\texttt{check}(\rho.f)} & S \cdot \langle \sigma, \texttt{skip} \rangle & \text{if } \sigma(x) = \rho & [\text{E-ChkField}] \\
\overline{D} \vdash S \cdot \langle \sigma, \texttt{check}(\{p_1, \ldots, p_n\}) \rangle & \longrightarrow^{t\,:\,\epsilon} & S \cdot \langle \sigma, \texttt{check}(\{p_1\}; \ldots; \texttt{check}(\{p_n\})) \rangle & & [\text{E-ChkSet}] \\
\overline{D} \vdash S \cdot \langle \sigma, \texttt{check}(x[e_1..e_2 : e_3]) \rangle & \longrightarrow^{t\,:\,\epsilon} & S \cdot \langle \sigma, \texttt{skip} \rangle & \text{if } \sigma(e_1) \geq \sigma(e_2) & [\text{E-ChkEmpty}] \\
\overline{D} \vdash S \cdot \langle \sigma, \texttt{check}(x[e_1..e_2 : e_3]) \rangle & \longrightarrow^{t\,:\,\texttt{check}(\rho[i])} & S \cdot \langle \sigma, \texttt{check}(\{x[(e_1 + e_3)..e_2 : e_3]\}) \rangle & \text{if } \rho = \sigma(x), i = \sigma(e_1), i < \sigma(e_2) & [\text{E-ChkIndex}] \\
\overline{D} \vdash S \cdot \langle \sigma, s_1; s_2 \rangle & \longrightarrow^{a} & S' \cdot \langle \sigma', s_1'; s_2 \rangle & \text{if } S \cdot \langle \sigma, s_1 \rangle \longrightarrow^{a} S' \cdot \langle \sigma', s_1' \rangle & [\text{E-Seq}] \\
\overline{D} \vdash S \cdot \langle \sigma, \texttt{skip}; s \rangle & \longrightarrow^{t\,:\,\epsilon} & S \cdot \langle \sigma, s \rangle & & [\text{E-Seq2}] \\
\overline{D} \vdash S \cdot \langle \sigma, \texttt{if } be\ s_1\ s_2 \rangle & \longrightarrow^{t\,:\,\epsilon} & S \cdot \langle \sigma, s_1 \rangle & \text{if } \sigma(be) = \texttt{true} & [\text{E-If}] \\
\overline{D} \vdash S \cdot \langle \sigma, \texttt{if } be\ s_1\ s_2 \rangle & \longrightarrow^{t\,:\,\epsilon} & S \cdot \langle \sigma, s_2 \rangle & \text{if } \sigma(be) = \texttt{false} & [\text{E-If2}] \\
\overline{D} \vdash S \cdot \langle \sigma, L \rangle & \longrightarrow^{t\,:\,\epsilon} & S \cdot \langle \sigma, s_1;\ \texttt{if } be\ \texttt{skip}\ \{s_2; L\} \rangle & L = \texttt{loop\{}\ s_1;\ \{\ \texttt{if } be\ \texttt{break}\ \};\ s_2\ \texttt{\}} & [\text{E-Loop}] \\
\overline{D} \vdash S \cdot \langle \sigma, x\ \texttt{=}\ y.f \rangle & \longrightarrow^{t\,:\,\texttt{acc}(\rho.f)} & S \cdot \langle \sigma[x := v], \texttt{skip} \rangle & \text{if } \sigma(y) = \rho \text{ and } S(\rho.f) = v & [\text{E-Read}] \\
\overline{D} \vdash S \cdot \langle \sigma, y.f\ \texttt{=}\ x \rangle & \longrightarrow^{t\,:\,\texttt{acc}(\rho.f)} & S[\rho.f := v] \cdot \langle \sigma, \texttt{skip} \rangle & \text{if } \sigma(y) = \rho \text{ and } \sigma(x) = v & [\text{E-Write}] \\
\overline{D} \vdash S \cdot \langle \sigma, x\ \texttt{=}\ y[i] \rangle & \longrightarrow^{t\,:\,\texttt{acc}(\rho[i])} & S \cdot \langle \sigma[x := v], \texttt{skip} \rangle & \text{if } \sigma(y) = \rho \text{ and } S(\rho[i]) = v & [\text{E-ARead}] \\
\overline{D} \vdash S \cdot \langle \sigma, y[i]\ \texttt{=}\ x \rangle & \longrightarrow^{t\,:\,\texttt{acc}(\rho[i])} & S[\rho[i] := v] \cdot \langle \sigma, \texttt{skip} \rangle & \text{if } \sigma(y) = \rho \text{ and } \sigma(x) = v & [\text{E-AWrite}] \\
\overline{D} \vdash S \cdot \langle \sigma, \texttt{acq}(x) \rangle & \longrightarrow^{t\,:\,\texttt{acq}(\rho)} & S[\rho := t] \cdot \langle \sigma, \texttt{skip} \rangle & \text{if } \sigma(x) = \rho \text{ and } S(\rho) = \perp & [\text{E-Acq}] \\
\overline{D} \vdash S \cdot \langle \sigma, \texttt{rel}(x) \rangle & \longrightarrow^{t\,:\,\texttt{rel}(\rho)} & S[\rho := \perp] \cdot \langle \sigma, \texttt{skip} \rangle & \text{if } \sigma(x) = \rho \text{ and } S(\rho) = t & [\text{E-Rel}] \\
\overline{D} \vdash S \cdot \langle \sigma, x\ \texttt{= new}\ c \rangle & \longrightarrow^{t\,:\,\epsilon} & S \cdot \langle \sigma[x := \rho], \texttt{skip} \rangle & \text{if } \rho \text{ is fresh} & [\text{E-New}] \\
\overline{D} \vdash S \cdot \langle \sigma, x\ \texttt{= new_array}\ z \rangle & \longrightarrow^{t\,:\,\epsilon} & S \cdot \langle \sigma[x := \rho], \texttt{skip} \rangle & \text{if } \rho \text{ is fresh} & [\text{E-ANew}] \\
\overline{D} \vdash S \cdot \langle \sigma, x\ \texttt{=}\ e \rangle & \longrightarrow^{t\,:\,\epsilon} & S \cdot \langle \sigma[x := v], \texttt{skip} \rangle & \text{if } \sigma(e) = v & [\text{E-Assign}] \\
\overline{D} \vdash S \cdot \langle \sigma, x \leftarrow y \rangle & \longrightarrow^{t\,:\,\epsilon} & S \cdot \langle \sigma[x := \sigma(y)], \texttt{skip} \rangle & & [\text{E-Rename}] \\
\overline{D} \vdash S \cdot \langle \sigma, x\ \texttt{=}\ y.m(\overline{z}) \rangle & \longrightarrow^{t\,:\,\epsilon} & S \cdot \langle \sigma, \theta(s) \rangle & & [\text{E-Call}] \\
 & & & \text{if } m(\overline{z'})\ \{\ s; \texttt{return}\ r\ \} \in \overline{D} & \\
 & & & \text{and } \theta \text{ maps } FV(s) \setminus \{r\} \text{ to fresh names} & \\
 & & & \text{and } \theta \text{ maps } \overline{z'}, \texttt{this}, r \text{ to } \overline{z}, y, x, \text{ respectively.} &
\end{array}
$$

Figure 3.9: Semantics and Runtime Values for BFJ.

3.9.2 Data Races and Check Races

The *happens-before relation* $<_\alpha$ for a trace α is the smallest transitively-closed relation over the operations in α such that the relation $a <_\alpha b$ holds whenever a occurs before b in α and one of the following holds:

- Program order: The two operations performed by the same thread.
- Locking: The two operations acquire or release the same lock.

We introduce the following definitions:

- Two operations are *concurrent* if they are not ordered by happens before.
- Two accesses *conflict* if they access the same location.
- Two checks *conflict* if they check the same location.
- A trace has a *data race* on a location l if it has a pair of conflicting concurrent accesses to l.
- A trace has a *check race* on a location l if it has a pair of conflicting concurrent checks on l.
- A check $c = t$:`check(`l`)` *covers* an access $a = t$:`acc(`l`)` if:
  - c precedes a with no intervening t:`rel(`l`)`.
  - c succeeds a with no intervening t:`acq(`l`)`.
- A check $c = t$:`check(`l`)` is *legitimate* for an access $a = t$:`acc(`l`)` if:
  - c precedes a with no intervening t:`acq(`l`)`.
  - c succeeds a with no intervening t:`rel(`l`)`.
- A trace α has *precise* checks if each access has a covering check and each check is legitimate for some access.

We start with two technical lemmas that show how the notions of covering and legitimate checks constrain the happens-before relation for a trace.

Lemma 1. *If a trace α has an access a with a covering check c then for any action d by a different thread in α we have that:*

1. $c <_\alpha d \Rightarrow a <_\alpha d$
2. $d <_\alpha c \Rightarrow d <_\alpha a$

Proof. The check c can become either before or after the access in α.

- Case Before:

  $\alpha = \alpha_1.c.\alpha_2.a.\alpha_3$, where α_2 has no releases by thread t since check c covers a.

  Program order then shows that $d <_\alpha c \Rightarrow d <_\alpha a$.

  Since α_2 does not contain a release by thread t, $c <_\alpha d \Rightarrow a <_\alpha d$.

- Case After:

  $\alpha = \alpha_1.a.\alpha_2.c.\alpha_3$, where α_2 has no acquires by t since check c covers a

  By program order $c <_\alpha d \Rightarrow a <_\alpha d$.

  Since α_2 does not contain an acquire by thread t, $d <_\alpha c \Rightarrow d <_\alpha a$.

□

Lemma 2. *If a trace α has a check c that is legitimate for an access a then for any action d by a different thread in α we have:*

1. $a <_\alpha d \Rightarrow c <_\alpha d$
2. $d <_\alpha a \Rightarrow d <_\alpha c$

Proof. The proof is similar to the above lemma. □

We next show that the notion of covering checks guarantees no missed races (false negatives), and the notion of legitimate checks guarantees no false alarms (false positives).

Lemma 3. *Let l be a location and suppose each access to l in α has a covering check. If α has no check race on l then α has no data race on l.*

Proof. Let $t\!:\!\texttt{acc}(l)$ and $u\!:\!\texttt{acc}(l)$ be two accesses in α whose covering checks are not racy. Without loss of generality we assume $t:\texttt{check}(l) <_\alpha u:\texttt{check}(l)$. By Lemma 1(1), $t\!:\!\texttt{acc}(l) <_\alpha u\!:\!\texttt{check}(l)$, and hence by Lemma 1(2) $t\!:\!\texttt{acc}(l) <_\alpha u\!:\!\texttt{acc}(l)$, so the accesses are race-free. □

Lemma 4. *Let l be a location and suppose each check in α on l is legitimate for some access. If α has a check race on l then α has a data race on l.*

Proof. Suppose α has two race-free accesses $t\!:\!\texttt{acc}(l)$ and $u\!:\!\texttt{acc}(l)$, where $t\!:\!\texttt{acc}(l) <_\alpha u\!:\!\texttt{acc}(l)$. Each access has a covering check $t:\texttt{check}(l)$ and $u:\texttt{check}(l)$. By Lemma 2(1), $t\!:\!\texttt{check}(l) <_\alpha u\!:\!\texttt{acc}(l)$. By Lemma 2(2), $t\!:\!\texttt{check}(l) <_\alpha u\!:\!\texttt{check}(l)$. □

By combining these ideas of covering and legitimate checks, we prove that a trace with precise checks has a check race if and only if the trace has a data race (and therefore running a dynamic race detector with these checks will report a race if and only if there is a data race).

Theorem 3.9.1. *Suppose α has precise checks. Then for all locations l, α has a check race on l if and only if α has a data race on l.*

Proof. By the application of Lemma 3 and 4. □

3.9.3 GoodChecks Judgment

Studies of type systems typically separate the problems of type inference and type checking. In our setting, we also separate the problems of inferring where to place checks and verifying that check placement is precise. BigFoot's check placement judgment shown in Figure 3.7 performs the former; the "good checks" judgment presented in this section performs the later. We refer to those judgments as CheckPlacement and GoodChecks, respectively.

The GoodChecks rules shown in Figure 3.10 include a subsumption rule [CC-Sub], and so it is not a syntax-directed algorithm like CheckPlacement; instead it is a mathematical definition designed to satisfy the usual preservation property plus other correctness properties discussed below regarding precise race detection.

The CheckPlacement algorithm uses both a history context H and anticipated context A to represent the forwards and backwards analysis. GoodChecks combines these two to form a single context $\Pi = H \cup A$. We define the entailment relation $\Pi \vdash h$ from a context $\Pi = H \cup A$ as simply $H \vdash h$. The GoodChecks rules are defined as follows:

- [CC-Skip], [CC-New], and [CC-ANew] do not change the context and always succeed.

- [CC-Assign] adds a new constraint representing the assignment to the post-context.

- [CC-Chk] adds the checked paths ($C^{\checkmark}$) to the post context. It only succeeds if each path p to check has already been accessed. This condition prevents false positives by preventing checking locations which have not yet been accessed. We always delay checks and never bring them forward so a location must have been accessed in order to be checked.

- [CC-Read] removes any anticipated accesses to $y.f$ and adds an access to $y.f$. Removing the anticipated access is safe because we are adding in an access to the same location.

- [CC-Write] also removes any anticipated accesses to $y.f$ and adds in an access to $y.f$.

- [CC-ARead] and [CC-AWrite] are similar to the above.

- [CC-Seq] allows for the chaining of two statements with the post-context of the first becoming the pre-context of the second.

- [CC-If] checks both the then and the else branches using the pre-context along with the information gained about *be*. The resulting post-contexts must match and are used for the post-context of the whole expression. Rule [CC-Sub] below can be used to bring the two post-contexts into alignment.

- [CC-Loop] enters the loop with a context of Π_{inv}. Statement s_1 is checked with this context and produces a new context $\Pi_1 \cup \{\neg be\}$. Statement s_2 is then checked with $\Pi_1 \cup \{\neg be\}$ and produces the post context Π_{inv} which can safely check s_1. Upon exiting the loop s_1 has run and the *be* is true so the resulting post-context is $\Pi_1 \cup \{be\}$.

- [CC-Acq] does not change the context. However, it does check that all accesses in the context have already been checked (as checking them after acquiring the lock may cause a false negative). It also checks that there are no anticipated accesses in the context as in the backward analysis the anticipated accesses can not be safely moved before an acquire.

- [CC-Rel] removes all history information from the context except boolean expressions. We must remove accesses and checks as the checks made so far are only valid while the lock about to be released is held. For every variable that has been accessed but not checked yet there must be an anticipated access in the post context. This constraint allows the delaying of checks outside of critical sections but only when a later access can be guaranteed.

- [CC-Call] does not modify the context and requires that it not contain any items in the kill set of that method. All accesses which may be killed in the method must be checked before the call. We can use [CC-Sub], shown below, to remove checked access that may be killed but we can not use subsumption to remove unchecked accesses so the analysis remains sound.

- [CC-Sub] allows us to conservatively approximate contexts according to the following context ordering:

$$(H_1 \cup A_1) \preceq (H_2 \cup A_2) \quad \text{iff} \quad \begin{cases} H_1 \sqsupseteq H_2 \\ H_1 \vdash A_1 \sqsubseteq A_2 \\ \forall p^{\lhd} \in H_1.\ (H_2 \vdash p^{\lhd}) \vee (H_1 \vdash p^{\checkmark}) \vee (H_2 \bullet A_2 \vdash p^{\diamond}) \end{cases}$$

Run-Time States We introduce the rules shown in Figure 3.11 to extend the GoodChecks relation to run-time states. The judgment $\sigma; \alpha \Vdash_t h$ determines when a history property h holds in a given thread-local store σ of thread t. The execution history α is used to validate past checks and accesses in h. The judgment $\sigma; \alpha \Vdash_t \Pi$ extends the previous judgment to contexts, and ensures (via C1) each check in α has a legitimizing previous access in α, and also each access in α either 1) has a covering check in α (via A1 or A2) or 2) the context Π records a corresponding past or anticipated access (via A3).

We extend the store σ to map paths (used by the static analysis) to sets of locations (used by the dynamic semantics) as follows:

$$\begin{array}{lcll} \sigma(x.f) & = & \{\ \sigma(x).f\ \} & \\ \sigma(x[e_1..e_2:e_3]) & = & \{\ p[j] : & p = \sigma(x), \\ & & & j = \sigma(e_1) + i\ \sigma(e_3), \\ & & & \sigma(e_1) \leq j < \sigma(e_2) \quad \} \end{array}$$

The rules [CC-Thread] and [CC-State] then extend this well-formedness criteria to threads and states, respectively. Note that [CC-State] does not constraint the heap S in any way, since we do not reason about heap contents statically. If we were to, for example, include alias assumptions in our core analysis, then we would need to ensure that all of our alias assumptions are true for S.

We assume $x \notin \mathit{Vars}(\Pi)$ in [CC-New], [CC-Assign], [CC-Read], [CC-ARead], [CC-ANew], [CC-Rename], [CC-Call].

$$\boxed{\Vdash s : \Pi \to \Pi'}$$

[CC-Skip]
$$\frac{}{\Vdash \texttt{skip} : \Pi \to \Pi}$$

[CC-Assign]
$$\frac{\Pi' = \Pi \cup \{x = e\}}{\Vdash x = e : \Pi \to \Pi'}$$

[CC-Rename]
$$\frac{\Pi' = H[y := x] \cup A}{\Vdash x \leftarrow y : H \cup A[x := y] \to \Pi'}$$

[CC-Chk]
$$\frac{\forall p \in C.\ \Pi \vdash p^{\lhd}}{\Vdash \texttt{check(}C\texttt{)} : \Pi \to \Pi \cup C^{\checkmark}}$$

[CC-New]
$$\frac{}{\Vdash x = \texttt{new}\ c : \Pi \to \Pi}$$

[CC-Read]
$$\frac{\Pi' = \Pi \setminus \{y.f^{\diamond}\} \cup \{y.f^{\lhd}\}}{\Vdash x = y.f : \Pi \to \Pi'}$$

[CC-Write]
$$\frac{\Pi' = \Pi \setminus \{y.f^{\diamond}\} \cup \{y.f^{\lhd}\}}{\Vdash y.f = x : \Pi \to \Pi'}$$

[CC-ANew]
$$\frac{}{\Vdash x = \texttt{new}\ c : \Pi \to \Pi}$$

[CC-ARead]
$$\frac{\Pi' = \Pi \setminus \{y[z]^{\diamond}\} \cup \{y[z]^{\lhd}\}}{\Vdash x = y[z] : \Pi \to \Pi'}$$

[CC-AWrite]
$$\frac{\Pi' = \Pi \setminus \{y[z]^{\diamond}\} \cup \{y[z]^{\lhd}\}}{\Vdash y[z] = x : \Pi \to \Pi'}$$

[CC-Seq]
$$\frac{\begin{array}{c}\Vdash s_1 : \Pi \to \Pi_1 \\ \Vdash s_2 : \Pi_1 \to \Pi_2\end{array}}{\Vdash s_1; s_2 : \Pi \to \Pi_2}$$

[CC-If]
$$\frac{\begin{array}{c}\Vdash s_1 : \Pi \cup \{be\} \to \Pi' \\ \Vdash s_1 : \Pi \cup \{\neg be\} \to \Pi'\end{array}}{\Vdash \texttt{if}\ be\ s_1\ s_2 : \Pi \to \Pi'}$$

[CC-Loop]
$$\frac{\begin{array}{c}\Vdash s_1 : \Pi_{inv} \to \Pi_1 \\ \Vdash s_2 : \Pi_1 \cup \{\neg be\} \to \Pi_{inv}\end{array}}{\Vdash \texttt{loop\{}\ s_1;\ \texttt{\{ if}\ be\ \texttt{break \}};\ s_2\ \texttt{\}} : \Pi_{\mathrm{inv}} \to \Pi_1 \cup \{be\}}$$

[CC-Acq]
$$\frac{\begin{array}{c}\forall p.\ p^{\diamond} \notin \Pi \\ \forall p.\ p^{\lhd} \in \Pi \Rightarrow \Pi \vdash p^{\checkmark}\end{array}}{\Vdash \texttt{acq(}x\texttt{)} : \Pi \to \Pi}$$

[CC-Rel]
$$\frac{\forall p.\ p^{\lhd} \notin \Pi \text{ and } p^{\checkmark} \notin \Pi}{\Vdash \texttt{rel(}x\texttt{)} : \Pi \to \Pi}$$

[CC-Call]
$$\frac{\begin{array}{c}\Pi \cap \mathit{KillSetHistory}(m) = \emptyset \\ \Pi \cap \mathit{KillSetAnticipated}(m) = \emptyset\end{array}}{\Vdash x = y.m(\overline{z}) : \Pi \to \Pi}$$

[CC-Sub]
$$\frac{\begin{array}{c}\Vdash s : \Pi_1' \to \Pi_2' \\ \Pi_1 \preceq \Pi_1' \\ \Pi_2' \preceq \Pi_2\end{array}}{\Vdash s : \Pi_1 \to \Pi_2}$$

$$\boxed{\Vdash \mathit{meth}}$$

[CC-Method]
$$\frac{\Vdash s}{\Vdash m(\overline{x})\{s; \texttt{return}\ z\}}$$

$$\boxed{\Vdash D}$$

[CC-Class]
$$\frac{\forall\ \mathit{meth} \in \overline{\mathit{meth}}.\ \Vdash \mathit{meth}}{\Vdash \texttt{class}\ c\{\overline{f}\ \overline{\mathit{meth}}\}}$$

$$\boxed{\Vdash s}$$

[CC-Stmt]
$$\frac{\Vdash s : \emptyset \to \emptyset}{\Vdash s}$$

$$\boxed{\Vdash \overline{D}\ \overline{s}}$$

[CC-Program]
$$\frac{\begin{array}{c}\forall D \in \overline{D}.\ \Vdash D \\ \forall i.\ \Vdash s_i\end{array}}{\Vdash \overline{D}\ s_1 \| \ldots \| s_n}$$

Figure 3.10: GoodChecks Rules.

$\boxed{\overline{D}; \alpha \Vdash \Sigma}$ $\boxed{\overline{D}; \alpha \Vdash_t \langle \sigma, s \rangle}$

[CC-State]

$$\frac{\forall D \in \overline{D}.\ \ \Vdash D \qquad \forall t \in \mathit{Tid}.\ \ \overline{D}; \alpha \Vdash_t T(t)}{\overline{D}; \alpha \Vdash S \cdot T}$$

[CC-Thread]

$$\frac{\sigma; \alpha \Vdash_t \Pi \qquad \Vdash s : \Pi \to \emptyset}{\overline{D}; \alpha \Vdash_t \langle \sigma, s \rangle}$$

$\boxed{\sigma; \alpha \Vdash_t \Pi}$

[CC-Context]

$$\frac{\begin{array}{lr} \forall h \in \Pi.\ \ \sigma; \alpha \Vdash_t h & \\ \text{Each } t : \texttt{check(}l\texttt{)} \text{ in } \alpha \text{ is preceded by } t : \texttt{acc(}l\texttt{)} \text{ with no intervening } t : \texttt{rel(\_)} & \text{(C1)} \\ \text{Each } t : \texttt{acc(}l\texttt{)} \text{ in } \alpha \text{ is} \begin{cases} \text{preceded by } t : \texttt{check(}l\texttt{)} \text{ with no intervening } t : \texttt{rel(\_)} \text{ or} \\ \text{followed by } t : \texttt{check(}l\texttt{)} \text{ with no intervening } t : \texttt{acq(\_)} \text{ or} \\ \text{not followed by } t : \texttt{acq(\_)} \text{ and } \exists p.\ (l \in \sigma(p) \text{ and } (p^{\diamond} \in \Pi \text{ or } p^{\lhd} \in \Pi)) \end{cases} & \begin{array}{r}\text{(A1)}\\ \text{(A2)}\\ \text{(A3)}\end{array} \end{array}}{\sigma; \alpha \Vdash_t \Pi}$$

$\boxed{\sigma; \alpha \Vdash_t h}$

[CC-BoolExp]

$$\frac{\sigma(be) = \texttt{true}}{\sigma; \alpha \Vdash_t be}$$

[CC-PastAccess]

$$\frac{\forall l \in \sigma(p).\ \begin{pmatrix} \alpha = \alpha_1.t : \texttt{acc(}l\texttt{)}.\alpha_2 \\ \alpha_2 \text{ does not contain } t : \texttt{rel(}\rho\texttt{)} \end{pmatrix}}{\sigma; \alpha \Vdash_t p^{\lhd}}$$

[CC-PastCheck]

$$\frac{\forall l \in \sigma(p).\ \begin{pmatrix} \alpha = \alpha_1.t : \texttt{check(}l\texttt{)}.\alpha_2 \\ \alpha_2 \text{ does not contain } t : \texttt{rel(}\rho\texttt{)} \end{pmatrix}}{\sigma; \alpha \Vdash_t p^{\checkmark}}$$

Figure 3.11: GoodChecks rules for Runtime States.

3.9.4 Correctness of CheckPlacement

Any program satisfying the CheckPlacement judgment will satisfy the GoodChecks judgment.

Lemma 5. *If* $\vdash \overline{D}\ s_1 \| \ldots \| s_n$ *then* $\Vdash \overline{D}\ s_1 \| \ldots \| s_n$.

Proof. Follows from Lemma 6. $\square$

Lemma 6. *If* $\vdash s$ *then* $\Vdash s$.

Proof. If $\vdash s$ then $s = s'; \texttt{check}(C)$ where $\vdash s' : \emptyset \bullet A \to H \bullet \emptyset$ and $C = \mathit{Checks}(H, \emptyset)$ from [Stmt]. Hence $\Vdash s' : A \to H$ by Lemma 7 so $\Vdash s : A \to H \cup C^{\surd}$ by [CC-Seq], and so by [CC-Sub] $\Vdash s : \emptyset \to \emptyset$, and hence $\Vdash s$. $\square$

Lemma 7. *If* $\vdash s : H \bullet A' \to H' \bullet A$ *then* $(\Vdash s : H \cup A' \to H' \cup A)$.

Proof. By induction on the derivation of $(\vdash s : H \bullet A' \to H' \bullet A)$ and case analysis on the rule concluding that derivation.

- [Assign] where $s = (x = e)$: In this case we have

  $$\vdash x = e : H \bullet A[x := e] \to H \cup \{x = e\} \bullet A$$

  where $x \notin \mathit{Vars}(e, H)$. Rule [CC-Assign] gives us

  $$\Vdash x = e : H \cup A[x := e] \to H \cup \{x = e\} \cup A[x := e]$$

  Finally, $H \cup A[x := e] \cup \{x = e\} \preceq H \cup \{x = e\} \cup A$ by our assumption about entailment, so by [CC-Sub]

  $$\Vdash x = e : H \cup A[x := e] \to H \cup \{x = e\} \cup A$$

  as required.

- [Rename] where $s = (x \leftarrow y)$: In this case we have

$$\vdash x \leftarrow y : H \bullet A[x := y] \rightarrow H[y := x] \bullet A$$

where $x \notin \mathit{Vars}(H)$. Rule [CC-Assign] gives us the desired

$$\Vdash x \leftarrow y : H \cup A[x := y] \rightarrow H[y := x] \cup A$$

- [Write] where $s = (y.f = x)$: In this case we have

$$\vdash y.f = x : H \bullet A \cup \{y.f^{\diamond}\} \rightarrow H \cup \{y.f^{\triangleleft}\} \bullet A$$

Rule [CC-Write] gives us

$$\Vdash y.f = x : H \cup A \cup \{y.f^{\diamond}\} \rightarrow H \cup A \cup \{y.f^{\triangleleft}\}$$

- [Read] where $s = (x = y.f)$: In this case we have

$$\vdash x = y.f : H \bullet A \setminus x \cup \{y.f^{\diamond}\} \rightarrow H \cup \{y.f^{\triangleleft}\} \bullet A$$

Rule [CC-Read] gives us

$$\Vdash x = y.f : H \cup A \setminus x \cup \{y.f^{\diamond}\} \rightarrow H \cup \{y.f^{\triangleleft}\} \cup A \setminus y.f^{\diamond}$$

Finally, $H \cup \{y.f^{\triangleleft}\} \cup A \setminus y.f^{\diamond} \preceq H \cup \{y.f^{\triangleleft}\} \cup A$ so by [CC-Sub] we reach our desired

$$\Vdash x = y.f : H \cup A \setminus x \cup \{y.f^{\diamond}\} \rightarrow H \cup \{y.f^{\triangleleft}\} \cup A$$

- [Skip] where $s = \texttt{skip}$: Skip does not modify or place restrictions on the context in either GoodChecks or CheckPlacement and so is trivial.

- [New] where $s = (x = \texttt{new}\ c)$: In this case we have

$$\vdash x = \texttt{new}\ c : H \bullet A \setminus x \to H \bullet A$$

where $x \notin \mathit{Vars}(H)$. Rule [CC-New] gives us

$$\Vdash x = \texttt{new}\ c : H \cup A \setminus x \to H \cup A \setminus x$$

Finally, $H \cup A \setminus x \preceq H \cup A$ so by [CC-Sub]

$$\Vdash x = \texttt{new}\ c : H \cup A \setminus x \to H \cup A$$

- [A-New], [A-Write], and [A-Read] follow the same proofs as [New], [Write], and [Read].

- [Acq] where $s = \texttt{check(}C\texttt{);acq(}x\texttt{)}$: In this case we have

$$\vdash \texttt{check(}C\texttt{);acq(}x\texttt{)} : H \bullet \emptyset \to H \cup C^{\surd} \bullet A$$

where $C = \mathit{Checks}(H, \emptyset)$. Rule [CC-Chk] gives us

$$\Vdash \texttt{check(}C\texttt{)} : H \to H \cup C^{\surd}$$

and requires that $\forall p \in C.\ H \vdash p^{\triangleleft}$ which holds by the construction of C. Next

[CC-ACQ] gives us

$$\Vdash \texttt{acq(}x\texttt{)} : H \cup C^{\checkmark} \to H \cup C^{\checkmark}$$

and requires that $\forall p.\ p^{\diamond} \notin H \cup C^{\checkmark}$, which is trivially true, and $\forall p.\ p^{\lhd} \in H \cup C^{\checkmark} \Rightarrow H \cup C^{\checkmark} \vdash p^{\checkmark}$ which is true by the construction of C. Finally, $H \cup C^{\checkmark} \preceq H \cup C^{\checkmark} \cup A$ and so by [CC-SUB] and [CC-SEQ] we have the desired

$$\Vdash \texttt{check(}C\texttt{)};\texttt{acq(}x\texttt{)} : H \to H \cup C^{\checkmark} \cup A$$

- [REL] where $s = \texttt{check(}C\texttt{)};\texttt{rel(}x\texttt{)}$: In this case we have

$$\vdash \texttt{check(}C\texttt{)};\texttt{rel(}x\texttt{)} : H \bullet A \to H \setminus \{\_^{\checkmark}, \_^{\lhd}\} \bullet A$$

where $C = \mathit{Checks}(H, A)$. GOODCHECKS gives us

$$\Vdash \texttt{check(}C\texttt{)} : H \cup A \to H \cup A \cup C^{\checkmark}$$

and requires that $\forall p \in C.\ H \vdash p^{\lhd}$ which it does by the construction of C. $H \cup A \cup C^{\checkmark} \preceq H \setminus \{\_^{\checkmark}, \_^{\lhd}\} \cup A$ because we will never remove an access $p^{\lhd}$ where $p^{\checkmark} \notin H \cup C^{\checkmark}$. Thus by [CC-SUB] we have that

$$\Vdash \texttt{check(}C\texttt{)} : H \cup A \to H \setminus \{\_^{\checkmark}, \_^{\lhd}\} \cup A$$

and by [CC-REL] and the fact that $\forall p.\ p^{\lhd} \notin H \setminus \{\_^{\checkmark}, \_^{\lhd}\}$ and $p^{\checkmark} \notin H \setminus \{\_^{\checkmark}, \_^{\lhd}\}$ we have

$$\Vdash \texttt{rel(}x\texttt{)} : H \setminus \{\_^{\checkmark}, \_^{\lhd}\} \cup A \to H \setminus \{\_^{\checkmark}, \_^{\lhd}\} \cup A$$

Finally, by [CC-Seq] we have the desired

$$\Vdash \texttt{check(}C\texttt{);rel(}x\texttt{)} : H \cup A \to H \setminus \{\_^{\surd}, \_^{\triangleleft}\} \cup A$$

- [If] where $s = \texttt{if}\ be\ (s_1;\texttt{check(}C_1\texttt{)})\ (s_2;\texttt{check(}C_2\texttt{)})$: In this case we have

$$\vdash \texttt{if}\ be\ (s_1;\texttt{check(}C_1\texttt{)})\ (s_2;\texttt{check(}C_2\texttt{)}) : H_{in} \bullet A_{in} \to H_{out} \bullet A_{out}$$
$$\vdash s_1 : H_{in} \cup \{be\} \bullet A_{in} \to H_1' \bullet A_{out}$$
$$\vdash s_2 : H_{in} \cup \{\neg be\} \bullet A_{in} \to H_2' \bullet A_{out}$$

$$\begin{array}{lcl} C_1 & = & Checks(H_1', H_1' \sqcap H_2', A_{out}) \\ C_2 & = & Checks(H_2', H_1' \sqcap H_2', A_{out}) \\ A_{in} & = & H_1 \bullet A_1 \sqcap H_2 \bullet A_2 \\ H_{out} & = & (H_1' \cup C_1^{\surd}) \sqcap (H_2' \cup C_2^{\surd}) \end{array}$$

In order to conclude

$$\Vdash \texttt{if}\ be\ (s_1;\texttt{check(}C_1\texttt{)})\ (s_2;\texttt{check(}C_2\texttt{)}) : H_{in} \cup A_{in} \to H_{out} \cup A_{out}$$

We need to show that

$$\Vdash s_1;\texttt{check(}C_1\texttt{)} : H_{in} \cup A_{in} \cup \{be\} \to H_{out} \cup A_{out}$$

$$\Vdash s_2;\texttt{check(}C_2\texttt{)} : H_{in} \cup A_{in} \cup \{\neg be\} \to H_{out} \cup A_{out}$$

By induction, [CC-Chk], and [CC-Seq] we have

$$\Vdash s_1;\texttt{check(}C_1\texttt{)} : H_{in} \cup A_{in} \cup \{be\} \to H_1' \cup C_1^{\surd} \cup A_{out}$$

$$\Vdash s_2; \texttt{check(}C_2\texttt{)} : H_{in} \cup A_{in} \cup \{\neg be\} \rightarrow H_2' \cup C_2^{\checkmark} \cup A_{out}$$

Finally by [CC-Sub]

$$\Vdash s_1; \texttt{check(}C_1\texttt{)} : H_{in} \cup A_{in} \cup \{be\} \rightarrow H_{out} \cup A_{out}$$

$$\Vdash s_2; \texttt{check(}C_2\texttt{)} : H_{in} \cup A_{in} \cup \{\neg be\} \rightarrow H_{out} \cup A_{out}$$

- [Seq] where $s = s_1; s_2$: This case holds trivial by induction.

- [Call] where $s = (\texttt{check(}C\texttt{)}; x = y.m(\overline{z}))$: In this case we have

$$\vdash \texttt{check(}C\texttt{)}; x = y.m(\overline{z}) : H \bullet A \rightarrow H' \bullet A'$$

such that

$$\begin{array}{rcl} C & = & \mathit{Checks}(H, H \setminus \mathit{KillSetHistory}(m), A) \\ & = & \{p : p^{\triangleleft} \in H, H \setminus \mathit{KillSetHistory}(m) \not\Vdash p^{\triangleleft}, H \bullet A \not\Vdash p^{\diamond}\} \\ H' & = & (H \cup C^{\checkmark}) \setminus \mathit{KillSetHistory}(m) \\ A & = & A' \setminus x \setminus \mathit{KillSetAnticipated}(m) \end{array}$$

By [CC-Check]

$$\Vdash \texttt{check(}C\texttt{)} : H \cup A \rightarrow H \cup C^{\checkmark} \cup A$$

By [CC-Call]

$$\Vdash x = y.m(\overline{z}) : H' \cup A \rightarrow H' \cup A$$

Also $H' \cup A \preceq H' \cup A'$.

Finally, we need to show

$$H \cup C^{\surd} \cup A \preceq H' \cup A = (H \cup C^{\surd}) \setminus \mathit{KillSetHistory}(m) \cup A$$

and in particular that

$$\forall p^{\lhd} \in H.\ H' \vdash p^{\lhd} \text{ or } H' \bullet A \vdash p^{\diamond}$$

If $C^{\surd} \not\vdash p^{\surd}$ then $H \setminus \mathit{KillSetHistory}(m) \vdash p^{\lhd}$ or $H \cup C^{\surd} \vdash p^{\surd}$ or $H' \bullet A \vdash p^{\diamond}$ and so the desired context ordering follows. Hence

$$\begin{aligned}
&\Vdash \texttt{check(}C\texttt{)} : H \cup A \rightarrow H \cup C^{\surd} \cup A \\
&H \cup C^{\surd} \cup A \preceq H' \cup A \\
&\Vdash x = y.m(\overline{z}) : H' \cup A \rightarrow H' \cup A \\
&H' \cup A \preceq H' \cup A'
\end{aligned}$$

- [Loop] where $s =$ `check(`C_{in}`);loop{` s_1; { `if` be `break` }; `check(`C_{back}`)` }: In this case we have the following:

$$\begin{aligned}
&\vdash s : H_{in} \bullet A_{in} \rightarrow H_{out} \bullet A_{inv} \\
&\vdash s_1 : H_{inv} \bullet A_{in} \rightarrow H \bullet A_{inv} \\
&H_{back} = H \cup \{\neg be\} \\
&H_{out} = H \cup \{be\} \\
&C_{in} = \mathit{Checks}(H_{in}, H_{inv}, A_{in}) \\
&H_{in} \cup C_{in}^{\surd} \sqsupseteq H_{inv} \\
&C_{back} = \mathit{Checks}(H_{back}, H_{inv}, A_{in}) \\
&H_{back} \cup C_{back}^{\surd} \sqsupseteq H_{inv}
\end{aligned}$$

By induction,

$$\Vdash s_1 : H_{inv} \cup A_{in} \rightarrow H \cup A_{inv}$$

Also, by [CC-Chk],

$$\Vdash \texttt{check(}C_{back}\texttt{)} : H_{back} \cup A_{inv} \rightarrow H_{back} \cup A_{inv} \cup C^{\checkmark}_{back}$$

Also, $H_{back} \cup A_{inv} \cup C^{\checkmark}_{back} \preceq H_{inv} \cup A_{in}$. Hence by [CC-Loop]

$$\Vdash \texttt{check(}C_{in}\texttt{);loop\{}\ s_1\texttt{;}\ \texttt{\{ if}\ \mathit{be}\ \texttt{break \};}\ \texttt{check(}C_{back}\texttt{)}\ \texttt{\}} : H_{inv} \cup A_{in} \rightarrow H_{out} \cup A_{inv}$$

This case concludes via [CC-Seq] and [CC-Sub] based on:

$$\begin{aligned}
&H_{out} \cup A_{inv} \preceq H_{out} \cup A_{out}\\
&\Vdash \texttt{check(}C_{in}\texttt{)} : H_{in} \cup A_{in} \rightarrow H_{in} \cup A_{in} \cup C^{\checkmark}_{in}\\
&H_{in} \cup A_{in} \cup C^{\checkmark}_{in} \preceq H_{inv} \cup A_{in}
\end{aligned}$$

□

Assumption 1 (Entailment Assumptions)**.** We rely on the following assumptions about the entailment relationship.

1. $H \bullet A \vdash p^{\diamond}$ is monotonic in H and A.
2. $H \vdash h$ is monotonic in H.
3. $\{x = e\} \bullet \{p^{\diamond}\} \vdash p[x := e]^{\diamond}$
4. $\{x = e\} \bullet \{p[x := e]^{\diamond}\} \vdash p^{\diamond}$
5. $\{h\} \vdash h$
6. $H \bullet A \vdash p^{\diamond}$ only depends on boolean expressions in H

7. $\{x[e_1]^{\checkmark}, x[(e_1 + e_3)..e_2 : e_3]^{\checkmark}\} \vdash x[e_1..e_2 : e_3]^{\checkmark}$

3.9.5 Correctness of GoodChecks

We now show that if a program has passed GoodChecks then the program generates traces that have precise checks.

Definition 3.9.1. *A state* Σ *is* terminated *if all threads are* `skip`*.*

Theorem 3.9.2 (Correctness of GoodChecks)**.** *If* $\overline{D}; \epsilon \Vdash \Sigma_0$ *and* $\overline{D} \Vdash \Sigma_0 \longrightarrow^{\alpha} \Sigma'$ *and* Σ' *is terminated then* α *has precise checks.*

Proof. By the Preservation Theorem, we have that $\overline{D}; \alpha \Vdash \Sigma'$ where $\Sigma' = S \cdot T$. Pick any thread t, and let $\langle \sigma, \texttt{skip} \rangle = T(t)$. By [CC-State] and [CC-Thread], $\overline{D}; \alpha \Vdash_t \langle \sigma, \texttt{skip} \rangle$ where $\sigma; \alpha \Vdash_t \Pi$ and $\Vdash \texttt{skip} : \Pi \to \emptyset$ for some $\Pi \preceq \emptyset$. Hence [CC-Context] implies that (from C1) each check by t has a preceding legitimizing access. Moreover, since $\Pi \preceq \emptyset$, from the definition of $\preceq$ we know that Π has no anticipated access, and $\forall p^{\lhd} \in \Pi$ we have that $\Pi \vdash p^{\surd}$.

Consider any access t:`acc(`l`)` in α, which must satisfy one of the antecedents A1, A2, A3 in [CC-Context]. If the access satisfies A1 or A2, then it clearly has a covering check. If the access satisfies A3, then, since Π has no anticipated accesses, $\exists p.\ \sigma(p) = l$ and $p^{\lhd} \in \Pi$. Then, from above, $\Pi \vdash p^{\surd}$, and so by [CC-PastCheck] α contains a check covering the access. Hence, α has precise checks. $\square$

In order to prove the above induction we must prove that evaluation preserves well-formed states.

Theorem 3.9.3 (Preservation)**.** *If* $\overline{D}; \alpha \Vdash \Sigma$ *and* $\Sigma \longrightarrow^{a} \Sigma'$ *then* $\overline{D}; (\alpha.a) \Vdash \Sigma'$*.*

Proof. Suppose the action a is performed by thread t. From the rule [CC-State] and the definition of our transition relation, we have:

$$\begin{array}{l}\Vdash D \qquad \forall D \in \overline{D} \\ \overline{D}; \alpha \Vdash_i T(i) \qquad \forall i \in \text{Tid} \\ \Sigma = S \cdot T[t := \langle \sigma, s \rangle] \\ \Sigma' = S' \cdot T[t := \langle \sigma', s' \rangle] \\ \overline{D} \vdash S \cdot \langle \sigma, s \rangle \longrightarrow^a S' \cdot \langle \sigma', s' \rangle \end{array}$$

In addition, for any thread $T(i) = \langle \sigma_i, s_i \rangle$, rule [CC-Thread] requires that there is a Π_i such that:

$$\begin{array}{l}\sigma_i; \alpha \Vdash_i \Pi_i \\ \Vdash s_i : \Pi_i \to \emptyset \end{array}$$

If $i \neq t$, then an inspection of [CC-Context] shows that $\sigma_i; \alpha.a \Vdash_i \Pi_i$, and hence $\overline{D}; (\alpha.a) \Vdash_i T(i)$. For thread t, from Lemma 8 below we have that there exists Π_3 such that

$$\begin{array}{l}\Vdash s' : \Pi_3 \to \emptyset \\ \sigma'; \alpha.a \Vdash_t \Pi_3 \end{array}$$

Hence $\overline{D}; (\alpha.a) \Vdash_t \langle \sigma', s' \rangle$. Finally, $\overline{D}; (\alpha.a) \Vdash \Sigma'$ then follows by rule [CC-State]. $\square$

To prove the above we must prove preservation for an individual thread step. Given a well formed thread state, if we take one step of evaluation the thread state remains well-formed.

Lemma 8 (Preservation for Statements). *If* $\forall D \in \overline{D}.\ \Vdash D$ *and* a *is an action by thread* t *and*

$$\begin{array}{l}\Vdash s : \Pi_1 \to \Pi_2 \\ \sigma; \alpha \Vdash_t \Pi_1 \\ \overline{D} \vdash S \cdot \langle \sigma, s \rangle \longrightarrow^a S' \cdot \langle \sigma', s' \rangle \end{array}$$

then there exist Π_3 *such that:*

$$\Vdash s' : \Pi_3 \rightarrow \Pi_2$$

$$\sigma'; (\alpha.a) \Vdash_t \Pi_3$$

Proof. By induction on the derivation of $\Vdash s : \Pi_1 \rightarrow \Pi_2$ and case analysis on the rule used to conclude that derivation.

- [CC-If] where $s = \texttt{if}\ be\ s_1\ s_2$: There are two cases:

  - if $\sigma(be) = \texttt{true}$:

$s' = s_1, \sigma' = \sigma, a = t : \epsilon$	Via Evaluation
Let $\Pi_3 = \Pi_1 \cup \{be\}$	
$\sigma; \alpha \Vdash_t \Pi_1$	Given
Need to show $\sigma; \alpha \Vdash_t \Pi_1 \cup \{be\}$	as $\sigma(be) = \texttt{true}$
Need to show $\Vdash s_1 : \Pi_1 \cup \{be\} \rightarrow \Pi_2$	Shown via [CC-If]

  - If $\sigma(be) = \texttt{false}$:

    The false case is similar.

- [CC-Rel] where $s = \texttt{rel(}x\texttt{)}$: In this case,

  $s = \texttt{rel(}x\texttt{)}$

  $\Pi_2 = \Pi_1$

  $\forall p.\ p^{\triangleleft} \notin \Pi_1, p^{\checkmark} \notin \Pi_1$

  $s' = \texttt{skip}$

  $a = t{:}\texttt{rel(}p\texttt{)}$

  $\sigma' = \sigma$

  Let $\Pi_3 = \Pi_1$. We have $\Vdash s' : \Pi_1 \rightarrow \Pi_1$ via [CC-Skip].

  Since Π_1 does not contain prior accesses or checks, it only contains boolean expressions, and so $\forall h \in \Pi_1$ from $\sigma; \alpha \Vdash_t h$ we can conclude $\sigma; \alpha.a \Vdash_t h$. Also, properties C1, A1, A2, A3 are not invalidated by adding a to α so we conclude $\sigma; \alpha.a \Vdash_t \Pi_1$ as required.

- [CC-Acq] where $s = \texttt{acq(}x\texttt{)}$: In this case,

$\Pi_2 = \Pi_1$	Via [CC-Acq]
$\forall p.\ p^{\diamond} \notin \Pi_1$	Via [CC-Acq]
$\forall p.\ p^{\triangleleft} \in \Pi_1 \Rightarrow \Pi_1 \Vdash p^{\checkmark}$	Via [CC-Acq]
$a = t : \texttt{acq}(\rho)$	Via Evaluation
$s' = \texttt{skip}, \sigma' = \sigma$	Via Evaluation
Let $\Pi_3 = \Pi_2$	
$\Vdash s' : \Pi_2 \to \Pi_2$	Shown via [CC-Skip]

We must show $\sigma; (\alpha.a) \Vdash_t \Pi_1$.

For all $h \in \Pi_1$, we have $\sigma; (\alpha) \Vdash_t h$ and hence $\sigma; (\alpha.a) \Vdash_t h$ as a is not a release.

Since Π_1 does not contain any anticipated access, the requirements C1, A2, A2, and A3 on α also hold for $\alpha.a$. Hence $\sigma; (\alpha.a) \Vdash_t \Pi_1$.

- [CC-Chk] where $s = \texttt{check}(C)$

  There are four evaluation rules for s.

  - [E-ChkField] where $s = \texttt{check}(\{x.f\})$.

    In this case,

$s' = \texttt{skip}, \sigma' = \sigma$	Via Evaluation
$a = t{:}\texttt{check}(\sigma(x).f)$	Via Evaluation
$\Pi_1 \vdash x.f^{\triangleleft}$	Via [CC-Chk]
$\Pi_2 = \Pi_1 \cup \{x.f^{\checkmark}\}$	Via [CC-Chk]
Let $\Pi_3 = \Pi_2$	
$\Vdash s' : \Pi_2 \to \Pi_2$	Via [CC-Skip]

    It remains to show $\sigma; \alpha.a \Vdash_t \Pi_2$.

    Clearly $\sigma; \alpha.a \Vdash_t x.f^{\checkmark}$ and so $\forall h \in \Pi_2.\ \sigma; \alpha.a \Vdash_t h$.

    Since we are adding $t{:}\texttt{check}(\sigma(x).f)$ to α, we need to show by C1 there was a $t{:}\texttt{acc}(\sigma(x).f)$ in α with no later release, which is already guaranteed by $\sigma; \alpha \Vdash_t x.f^{\triangleleft}$.

    Hence we conclude $\sigma; \alpha.a \Vdash_t \Pi_2$.

– [E-ChkSet] where $C = \{p_1, \ldots, p_n\}$.

In this case,

$s' = \texttt{check(}\{p_1\}\texttt{)}; \ldots; \texttt{check(}\{p_n\}\texttt{)}$	Via Evaluation
$\sigma' = \sigma$	Via Evaluation
$a = t : \epsilon$	Via Evaluation

Let $\Pi_3 = \Pi_1$

From $\Vdash s : \Pi_1 \to \Pi_2$ we clearly have $\Vdash s' : \Pi_1 \to \Pi_2$ and $\sigma; \alpha.a \Vdash_t \Pi_1$.

– [E-ChkEmpty] where $C = x[e_1..e_2 : e_3]$ and $\sigma(e_1) \geq \sigma(e_2)$.

We have

$s' = \texttt{skip}$	Via Evaluation
$\sigma' = \sigma$	Via Evaluation
$a = t : \epsilon$	Via Evaluation
$\Pi_2 = \Pi_1 \cup \{x[e_1..e_2 : e_3]^{\checkmark}\}$	Via [CC-Chk]
Let $\Pi_3 = \Pi_2$	
$\Vdash s' : \Pi_2 \to \Pi_2$	Via [CC-Skip]

We need to show $\sigma; \alpha.a \Vdash_t \Pi_2$, which reduces to showing $\sigma; \alpha.a \Vdash_t x[e_1..e_2 : e_3]^{\checkmark}$, which holds via [CC-PastCheck] as $\sigma(x[e_1..e_2 : e_3])$ is empty.

– [E-ChkIndex] where $C = \{x[e_1..e_2 : e_3]\}$.

We have

$\rho = \sigma(x)$	Via Evaluation
$i = \sigma(e_1)$	Via Evaluation
$i < \sigma(e_2)$	Via Evaluation
$s' = \texttt{check(}\{x[(e_1 + e_3)..e_2 : e_3]\}\texttt{)}$	Via Evaluation
$\sigma' = \sigma$	Via Evaluation
$a = t : \texttt{check(}\rho[i]\texttt{)}$	Via Evaluation
$\Pi_2 = \Pi_1 \cup \{x[e_1..e_2 : e_3]^{\checkmark}\}$	Via [CC-Chk]

Let $\Pi_3 = \Pi_1 \cup \{x[e_1]^{\checkmark}\}$

Clearly $\sigma; \alpha.a \Vdash_t x[e_1]^{\checkmark}$.

Since we are adding check a to the trace, we need to show by C1 there was a corresponding access $t\!:\!\texttt{acc(}\rho[i]\texttt{)}$ in α with no later release, which is already guaranteed by $\sigma; \alpha \Vdash_t x[e_1..e_2 : e_3]^{\triangleleft}$.

Hence we conclude $\sigma; \alpha.a \Vdash_t \Pi_2$.

Also, we have $\Vdash s' : \Pi_1 \cup \{x[e_1]^{\checkmark}\} \to \Pi_1 \cup \{x[e_1]^{\checkmark}\} \cup \{x[e_1 + e_3..e_2 : e_3]^{\checkmark}\}$ via [CC-Chk], which via [CC-Sub] gives $\Vdash s' : \Pi_3 \to \Pi_2$ as required.

- [CC-Call] where $s = (x = y.m(\overline{z}))$: In this case,

$\overline{D} \Vdash m(\overline{x})\{s_m;\ \texttt{return}\ r\}$	Via [CC-Call]
$x \notin \mathit{Vars}(\Pi_1, e)$	
$s' = \theta(s_m)$	Via Evaluation
$\theta = \{\overline{z' := z}, \texttt{this} := y, r := x\}$	Via [CC-Call]
$\Pi \cap \mathit{KillSetHistory}(m) = \emptyset$	Via [CC-Call]
$\Pi \cap \mathit{KillSetAnticipated}(m) = \emptyset$	Via [CC-Call]
$\sigma' = \sigma, a = t : \epsilon$	Via Evaluation
Let $\Pi_3 = \Pi_1 = \Pi_2 = \Pi$	
Need to show $\sigma; \alpha.a \Vdash_t \Pi$	Given
Need to show $\Vdash s_m : \Pi \to \Pi$	
$\Vdash s_m : \emptyset \to \emptyset$	Via [CC-Method] and [CC-Stmt]
$\Vdash \theta(s_m) : \emptyset \to \emptyset$	Via Lemma 10
$\Vdash s_m : \emptyset \cup \Pi \to \emptyset \cup \Pi$	Via Lemma 9

- [CC-Loop] where $s = L$:

$L = \texttt{loop}\{\ s_1;\ \{\ \texttt{if}\ \mathit{be}\ \texttt{break}\ \};\ s_2\ \}$	
$s' = s_1; \texttt{if}\ \mathit{be}\ \texttt{skip}\ \{s_2; L\}$	Via Evaluation
$\sigma' = \sigma, a = t : \epsilon$	Via Evaluation
$\Pi' = \Pi_2 \setminus \{\mathit{be}\}$	
$\Vdash s_1 : \Pi_1 \to \Pi'$	Via [CC-Loop]
$\Vdash s_2 : \Pi' \cup \{\neg \mathit{be}\} \to \Pi_1$	Via [CC-Loop]
Let $\Pi_3 = \Pi_1$	
Need to show $\sigma; \alpha.a \Vdash_t \Pi_1$	Given
$\Pi_2 = \Pi_1 \cup \{\mathit{be}\}$	Via [CC-Loop]
Need to show $\Vdash s' : \Pi_1 \to \Pi' \cup \{\mathit{be}\}$	
$\Vdash L : \Pi_1 \to \Pi' \cup \{\mathit{be}\}$	Given
$\Vdash s_2; L : \Pi' \cup \{\neg \mathit{be}\} \to \Pi' \cup \{\mathit{be}\}$	Via [CC-Seq]
$\Vdash \texttt{skip} : \Pi' \cup \{\mathit{be}\} \to \Pi' \cup \{\mathit{be}\}$	Via [CC-Skip]
$\Vdash \texttt{if}\ \mathit{be}\ \texttt{skip}\ (s_2; L) : \Pi' \to \Pi' \cup \{\neg \mathit{be}\}$	Via [CC-Seq] and [CC-Sub]
$\Vdash s' : \Pi_1 \to \Pi' \cup \{\mathit{be}\}$	Via [CC-Seq]

- [CC-Seq] where $s = s1; s2$: There are two cases:

  - If $s_1 = \texttt{skip}$:

$S' = s_2, \sigma' = \sigma, a = t : \epsilon$	Via Evaluation
Let $\Pi_3 = \Pi_1$	
Need to show $\Vdash s_2 : \Pi_1 \to \Pi_2$	Shown via [CC-Seq]
Need to show $\sigma; \alpha \Vdash_t \Pi_1$	Given

  - If $s_1 \neq \texttt{skip}$:

$\overline{D} \vdash S \cdot \langle \sigma, s_1 \rangle \longrightarrow^a S' \cdot \langle \sigma', s_1' \rangle$	Via Evaluation
$\Vdash s_1 : \Pi_1 \to \Pi'$	Via [CC-Seq]
$\Vdash s_2 : \Pi' \to \Pi_2$	Via [CC-Seq]
$\sigma; \alpha \Vdash_t \Pi_1$	Given
$\Vdash s_1' : \Pi_4 \to \Pi'$	Via inductive hypothesis
$\sigma; (\alpha.a) \Vdash_t \Pi_4$	Via inductive hypothesis
Let $\Pi_3 = \Pi_4$	
Need to show $\sigma; (\alpha.a) \Vdash_t \Pi_4$	Shown via induction above
Need to show $\Vdash s_1'; s_2 : \Pi_3 \to \Pi_2$	Shown via application of [CC-Seq]

- [CC-Read] where $s = (x = y.f)$:

$\Pi_2 = (\Pi_1 \setminus \{y.f^{\diamond}\}) \cup \{y.f^{\lhd}\}$	Via [CC-Read]
$x \notin \mathit{Vars}(\Pi_1, e)$	
$s' = \texttt{skip}, \sigma' = \sigma[x := v], v = S(\sigma(y.f))$	Via Evaluation
Let $\Pi_3 = \Pi_2$	
Need to show $\Vdash s' : \Pi_2 \to \Pi_2$	Shown via [CC-Skip]
$a = t : \texttt{acc}(y.f)$	Via Evaluation
Need to show $\sigma'; (\alpha.a) \Vdash_t (\Pi_1 \setminus \{y.f^{\diamond}\}) \cup \{y.f^{\lhd}\})$	

  All actions in α proved by $A3$ are still proved because we have removed $y.f^{\diamond}$ but added in a $y.f^{\lhd}$. Clearly $\sigma; \alpha.a \Vdash_t y.f^{\lhd}$. a is proved by $A3$ because $y.f^{\lhd} \in \Pi_3$. All history properties in Π_1 remain proved in σ' as $x \notin \mathit{Vars}(\Pi_1)$.

- [CC-ARead] where $s = (x = y[z])$: The proof is similar to above.

- [CC-Write] where $s = (y.f = x)$:

$\Pi_2 = \Pi_1 \setminus \{f, y.f^{\diamond}\} \cup \{y.f^{\lhd}, x = y.f\}$	Via [CC-Write]
$s' = \texttt{skip}, \sigma' = \sigma$	Via Evaluation
Let $\Pi_3 = \Pi_2$	
Need to show $\Vdash s' : \Pi_2 \to \Pi_2$	shown via [CC-Skip]
$a = t : \texttt{acc}(y.f)$	Via Evaluation
Need to show $\sigma'; (\alpha.a) \Vdash_t \Pi_2$	

All actions in α proved by $A3$ are still proved because we have removed $y.f^{\diamond}$ but added in a $y.f^{\lhd}$. Those proved by $A1$, $A2$, and $C1$ do not change because we have not changed α. Clearly $\sigma; \alpha.a \Vdash_t y.f^{\lhd}$. a is proved by $A3$ because $y.f^{\lhd} \in \Pi_3$.

- [CC-AWrite] where $s = (y[z] = x)$: The proof is similar to above.

- [CC-New] where $s = (x = \texttt{new}\ c)$:

$\Pi_2 = \Pi_1$	[CC-New]
$x \notin \mathit{Vars}(\Pi_1, e)$	
$\sigma' = \sigma[x := \rho]$	Via Evaluation
$s' = \texttt{skip}, a = t : \epsilon$	Via Evaluation
Let $\Pi_3 = \Pi_2$	
Need to show $\Vdash s' : \Pi_2 \to \Pi_2$	Shown via [CC-Skip]
Need to show $\sigma'; \alpha \Vdash_t \Pi_1$	Given

- [CC-ANew]: The proof is similar to above.

- [CC-Assign] where $s = (x = e)$:

$s' = \texttt{skip}, \sigma' = \sigma[x := v], a = t : \epsilon, v = \sigma(e)$	Via Evaluation
$x \notin \mathit{Vars}(\Pi_1, e)$	
$\Pi_2 = \Pi_1 \cup \{x = e\}$	Via [CC-Assign]
Let $\Pi_3 = \Pi_2$	
Need to show $\Vdash s' : \Pi_2 \to \Pi_2$	Shown via [CC-Skip]
Need to show $\sigma'; \alpha \Vdash_t \Pi_1 \cup \{x = e\}$	Shown via what is given and the new constraint is true based on σ'

- [CC-Rename] where $s = (x \leftarrow y)$:

$s' = \texttt{skip}, \sigma' = \sigma[x := \sigma(y)], a = t : \epsilon$	Via Evaluation
$x \notin \mathit{Vars}(\Pi_1)$	
$\Pi_1 = H \cup A[x := y]$	
$\Pi_2 = H[y := x] \cup A$	Via [CC-Rename]
Let $\Pi_3 = \Pi_2$	
Need to show $\Vdash s' : \Pi_2 \to \Pi_2$	Shown via [CC-Skip]
Need to show $\sigma'; \alpha \Vdash_t \Pi_2$	Shown via $x \notin \mathit{Vars}(\Pi_1)$ and $\sigma(y) = \sigma'(x)$

□

We now state several technical lemmas used in the arguments above. We extend the functions *KillSetAnticipated* and *KillSetHistory* from method names to statements in the expected manner.

Lemma 9 (Extension). *If* $\Vdash s : \Pi_1 \to \Pi_2$ *and* $\Pi' = \Pi \setminus \mathit{KillSetHistory}(s) \setminus \mathit{KillSetAnticipated}(s)$ *then* $\Vdash s : \Pi_1 \cup \Pi' \to \Pi_2 \cup \Pi'$.

Proof. By induction on the derivation of $\Vdash s : \Pi_1 \to \Pi_2$ and case analysis on the rule used to conclude that derivation.

- [CC-Skip], [CC-New], [CC-Assign]: These rules have no constraints on their input and output Π so the proof holds trivially.
- [CC-Acq]:

$\Pi_1 = \Pi_2$	[CC-Acq]
$\forall p.\ p^{\diamond} \notin \Pi_1$	[CC-Acq]
$\forall p.\ p^{\lhd} \in \Pi_1 \Rightarrow \Pi_1 \Vdash p^{\checkmark}$	[CC-Acq]
Need to show $\forall p.\ p^{\diamond} \notin (\Pi_1 \cup \Pi')$	[CC-Acq]
$\forall p.\ p^{\diamond} \notin \Pi'$	Because an acquire kills $p^{\diamond}$
Need to show $\forall p.\ p^{\lhd} \in (\Pi_1 \cup \Pi') \Rightarrow (\Pi_1 \cup \Pi') \Vdash p^{\checkmark}$	[CC-Acq]
$\forall p.\ p^{\lhd} \notin \Pi'$	Because an acquire kills $p^{\lhd}$

- [CC-Rel] where $s = \texttt{rel}(x)$: In this case $\Pi_1 = \Pi_2$ and $\forall p.\ p^{\lhd} \notin \Pi_1, p^{\checkmark} \notin \Pi_1$

  A release kills past checks and acquires, so $\forall p.\ p^{\lhd} \notin \Pi'$ and $p^{\checkmark} \notin \Pi'$. Hence $\Vdash s : \Pi_1 \cup \Pi' \to \Pi_2 \cup \Pi'$ as required.

- [CC-Read], [CC-Write], [CC-ARead], [CC-AWrite]: The only restriction is that the resulting Π_f must contain no $y.f^{\diamond}$ and must contain $y.f^{\lhd}$. Unioning with Π' can also not remove $y.f^{\lhd}$ so that condition is met. If Π' adds in a $y.f^{\diamond}$ then by [CC-Sub] the rule still holds as the resulting Π_f is greater then the original.
- [CC-If], [CC-Seq], [CC-Loop]: By induction.
- [CC-Call], [CC-Sub]: All requirements involving subterms are proved by induction. The only remaining requirements are proved because $\Pi_1 \preceq \Pi_2 \Rightarrow (\Pi_1 \cup \Pi) \preceq (\Pi_2 \cup \Pi)$.
- [CC-Check]: If $\Pi_1 \Vdash p^{\lhd}$ then $\Pi_1 \cup \Pi' \Vdash p^{\lhd}$.

□

Lemma 10 (Substitution)**.** *If* $\Vdash s : \Pi_1 \to \Pi_2$ *and* $\theta : \mathrm{Var} \to \mathrm{Var}$ *is a permutation on variables, then* $\Vdash \theta(s) : \theta(\Pi_1) \to \theta(\Pi_2)$*.*

Proof. Proof is by induction on the derivation of $\Vdash s : \Pi_1 \to \Pi_2$. □

3.9.6 Correctness of BigFoot

Theorem 3.9.4 (Correctness)**.** *Suppose* $P = \overline{D}\ s_1 \| \ldots \| s_n$ *is a program with inserted checks (i.e.* $\vdash P$*) that generates a trace* α *via*

$$\overline{D} \vdash \Sigma_0 \longrightarrow^{\alpha} \Sigma$$

where Σ_0 *is the initial state for* P *and* Σ *is terminated. Then* α *has a data race on a location* l *if and only if* α *has a check race on that location.*

Proof. By Lemma 5, we have that $\Vdash \overline{D}\ s_1 \| \ldots \| s_n$. Lemma 11 implies that $\overline{D}; \epsilon \Vdash \Sigma_0$. By Theorem 3.9.2, α has precise checks. Finally, Theorem 3.9.1 shows that α has a data race on location l if and only if it has a check race on l. □

Lemma 11. *If* $P = \overline{D}\ s_1 \| \ldots \| s_n$*,* $\Vdash P$*, and* $\Sigma_0 = S_0 \cdot T_0$ *is the initial state for* P*, then* $\overline{D}; \epsilon \Vdash \Sigma_0$*.*

Proof. By definition, T_0 maps each $t \in 1..n$ to $\langle \sigma, s_t \rangle$, where $Dom(\sigma) = FV(s_1) \cup \ldots \cup FV(s_n)$. Since $\Vdash P$ can only be derived via [CC-Prog], it must be that $\forall D \in \overline{D}.\ \Vdash D$. Also, we have that $\sigma; \epsilon \Vdash_t \emptyset$ via [CC-Ctxt]. Consider any t. Since $\vdash P$, we know that $\vdash s_t$ via [Program], which implies that $\Vdash s_t$ via Lemma 6. This is only derivable via rule [CC-Stmt], which means that $\Vdash s : \emptyset \to \emptyset$. From the above, rule [CC-Thread] allows us to conclude that $\overline{D}; \epsilon \Vdash_t \langle \sigma, s_t \rangle$. It then follows from rule [CC-State] that $\overline{D}; \epsilon \Vdash \Sigma_0$.

□

Chapter 4

Thread-Local Macro classifications

BigFoot's analysis relies mostly on access and synchronization information to remove redundant checks and coalesce needed checks. Our other analyses instead make use of high level reachability arguments to classify objects as race free. While all of heap memory is shared in Java many portions of memory may not be reachable by all threads. If a memory location is reachable by only a single thread then checks on that location can be safely removed. Our first macro analysis works with this idea and examines the correctness of eliding checks and lock operations on thread-local objects.

4.1 Introduction

Compilers often use escape analysis to elide locking operations on thread-local data. Similarly, dynamic race detectors may use escape analysis to elide race checks on thread-local data. In this section, we study the correctness of these two related optimization's when using a partial escape analysis, which identifies objects that are currently thread-local, but may later become thread-shared.

We show that lock elision based on partial escape analysis is unsound for the

Java memory model. We also show that race check elision based on a partial escape analysis weakens the precision of dynamic race detectors. Finally, we show that dynamic race detectors that use a partial escape analysis remain sound with respect to this weakened notion of precision.

When reasoning about heap allocated objects, compilers and other analyses must, in general, assume that concurrent threads can make arbitrary changes to any object. This uncertainty makes it difficult to reason about the possible behavior of code. For thread-local objects (that is, objects that are only accessible by a single thread), concurrent modifications are not possible until that object has *escaped* out of its allocating thread. An object escapes its allocating thread when it is assigned to a field of a thread-shared object. Many compilers and analyses make use of an *escape analysis* to determine which objects escape their allocating thread [34].

Escape analyses fall into two major categories. A *total escape analysis* determines if an object is *always* thread-local (i.e. it never escapes its allocating thread). On the other hand, a *partial escape analysis* determines if an object has not escaped its allocating thread *yet* [96].

Optimizing compilers, such as Hotspot [78], use an escape analysis to elide expensive synchronization operations on thread-local locks. We refer to lock elision based on total and partial escape analyses as *total* and *partial lock elision* respectively, and both have been proposed in the literature [96, 27].

Total lock elision has been proven sound [27]. However, lock elision optimizations have also been based on a partial escape analysis (see [96]). In this section, we show that partial lock elision is unsound in that it can introduce additional behaviors that are not permitted under the Java memory model.

Dynamic race detectors also leverage information about thread-local data. In part, a dynamic race detector typically performs a *race check* every time a thread of the target program reads or writes to an object. These race checks can be elided for access to thread-local objects.

We refer to race check elision based on total and partial escape analyses as *total* and *partial race check elision* respectively. We show that partial race check elision weakens the precision guarantees provided by a dynamic race detector. Partial race check elision never causes a race detector to miss the first data race in a program, but may cause it to miss subsequent data races.

Contributions: In summary, the contributions of this section are that we show:

- partial lock elision is unsound for compilers,
- partial race check elision may cause a race detector to miss some races in an execution,
- and partial race check elision will never cause a race detector to miss the first race in an execution.

4.2 Review of Data Races

Race detectors and other analyses use the concept of a trace to analyze a specific execution of a given program. We define a program trace α as a sequence of all the operations performed by the various threads. These operations include reads and writes of object fields as well as lock acquire and releases.

The happens-before relation $<_\alpha$ for a trace α is the smallest transitively-closed relation over the operations in the trace such that the relation $a <_\alpha b$ holds whenever a occurs before b in the trace and one of the following holds:

- Program order: The two operations are performed by the same thread.
- Locking: The two operations acquire or release the same lock.
- Fork: One operation forks a new thread and the other operation is by that new thread.

If two operations in a trace are not related by the happens-before relation, then they are considered *concurrent*. Two memory access *conflict* if they both access (read or write) the same address, and at least one of the operations is a write. Using this terminology, a trace has a *race condition* on a particular address if it has two concurrent conflicting accesses to that address.

Data races lead to unexpected behavior (in Java [91, 68]) or undefined behavior (in C++ [18]) and as such, compilers are not allowed to introduce data races when optimizing code. As we examined in section xxx the Java memory model uses a race-freedom implies sequential consistency standard. By introducing data races where none previously existed the Java compiler may break sequential consistency even when there was no data race in the original program.

Likewise, for a race detector to be *precise* it must not miss data races. More specifically, we say a dynamic race detector is

- *trace precise* if it correctly reports whether a program trace has a race, and
- *address precise* if it correctly reports all addresses in a trace that have race conditions.

We show that partial race check elision weakens the precision of a dynamic race detector from address precise to trace precise; in particular, one race in a trace may prevent subsequent races on different addresses from being detected. We prove, however, that the first race in a trace is always detected and so partial race check elision is still trace precise, which is sufficient for many applications. In particular a trace precise detector provides sufficient guarantees to reinstate sequential consistency, atomicity, and determinism.

4.3 Partial Lock Elision is Unsound

A partial escape analysis marks the point at which an object escapes its allocating thread. Partial lock elision uses this information to remove all acquire and release operations on the object before it escapes.

To illustrate why partial lock elision is unsound, consider the program shown in Figure 4.1. Thread 1 allocates a new `Ref` object, initializes its `f` field with `11` and then `12` inside a synchronized block, and then shares its address via `p.o` with thread 2. Thread 2 busy waits until `p.o` is non-`null` and then reads `f` inside a synchronized block. The two threads race on `p.o` due to the write from thread 1 and the busy wait from thread 2. However, the accesses to field `f` are race-free as all three accesses are protected by the lock created in thread 1 line 1. Because these accesses are race-free sequential consistency is maintained and Thread 2 can never read the partially-initialized value of `11`. Consequently, the assertion on line 14 never fails.

Because the synchronization on the lock at `r0` in Thread 1 happens before the lock escapes, partial lock elision as described in [96] would remove it, resulting in the code in Figure 4.1 after lock elision. In this new code, there is no happens before edge between the writes to `f` by Thread 1 and the read by Thread 2 because Thread 1's synchronization has been removed. Since these operations are now involved in a data race, the Java memory model allows either value, the first write of `11` or the second write of `12`, to be read by Thread 2 allowing the assertion to fail. Thus, the compiler has introduced a data race on `f` and an assertion violation that was not present in the original program. This behavior violates Java's memory model and breaks sequential consistency even when the user code contained no data races.

4.4 Partial Race Check Elision is not Address Precise

To illustrate why partial race check elision not address precise, consider the program in Figure 4.2. Here, Thread 1 creates a new `Ref` object, writes to `f`, and then shares the object by writing its address to `p.o`. Thread 2 reads the address of the `Ref` from `p.o` and writes to `f`. A trace of this program reveals two races: the first of which occurs on `p.o` on lines 31 and 32. Race checks are not elided on lines 31 and 32 because `p` is a shared object, thus this race is caught. However, there is a second race

```
            Before Lock Elision                              After Lock Elision
-----------------------------------------------------------------------------------------------
     Thread1             |     Thread2                 Thread1             |     Thread2
-------------------------|----------------         ------------------------|----------------
                         |                                                 |
Ref r0 = new Ref();      |                         Ref r0 = new Ref();     |
synchronized(r0){        |                                                 |
  r0.f = 11;             |                         r0.f = 11;              |
  r0.f = 12;             |                         r0.f = 12;              |
}                        |                                                 |
p.o = r0;                |                         p.o = r0;               |
                         | int r2;                                         | int r2;
                         | Ref r1 = null;                                  | Ref r1 = null;
                         | while(r1 == null)                               | while(r1 == null)
                         |   r1 = p.o;                                     |   r1 = p.o;
                         | synchronized(r1){                               | synchronized(r1){
                         |   r2 = r1.f;                                    |   r2 = r1.f;
                         | }                                               | }
                         | assert(r2 != 11);                               | assert(r2 != 11);
```

Figure 4.1: Example of Partial Lock Elision (`p` is a shared object and `p.o` is null initialized)

```
     Thread 1           |     Thread 2
------------------------|----------------
Ref r1 = new Ref();     |
r1.f = 11;              |
p.o = r1;               |
                        | Ref r1 = p.o;
                        | r1.f = 12;
```

Figure 4.2: Example of Partial Race Check Elision (`p` is a shared object)

in the program between the two writes to `f` on lines 30 and 33. In this case, the first race check on line 30 happens before `r1` has escaped and would be elided by partial race check elision. The check on line 33 is not elided because by this time the `Ref` has escaped. However, the race on `f` will not be detected due to the previously elided check. Note, this race is only missed due to the previous racy write to the shared pointer `p.o`, so the analysis is still trace precise for this particular trace.

4.5 Partial Race Check Elision is Trace Precise

We next show that partial race check elision is trace precise in general.

4.5.1 Idealized Language SimpleJava

We formalize our proof in terms of the idealized language SimpleJava shown in Figure 4.3. A programs P consists of a sequence class definitions D (containing methods and fields) as well as a main expression e. Expressions can create new objects (`new`), read from fields ($e.f$), assign to fields ($e.f = e$), create temporary variables (`let` $x = e$ `in` e), call methods ($e.m(\overline{e})$), acquire and release locks (`acq` e and `rel` e), and fork off new threads (`fork` e).

Figure 4.3 also shows the semantics for our language. A running program has a heap H and a thread set T. The heap maps locations to values and locks to the thread holding them. Values v are addresses p and `null`. A location $l = p.f$ is an object address p along with a field f. The thread set maps thread identifiers to expressions. This semantics is standard, but includes actions that are emitted for every evaluation step.

A program starts with an empty heap $\emptyset$, and a thread set $T = [t := e]$ with a single thread e with thread identifier t. A single evaluation step

$$P \vdash H, T \rightarrow^{a} H', T'$$

produces an action a. Taken in sequence these actions form a trace α. We include P in the evaluation relation to facilitate method lookup, and we assume method names are unique.

4.5.2 Partial Race Check Elision Algorithm

Figure 4.4 shows a partial race check elision algorithm. This algorithm performs a dynamic thread escape analysis, recording in G all addresses reachable by multiple threads. The judgement

$$P \vdash G, H, T \rightarrow^{a}_{b} G', H', T'$$

$$
\begin{array}{llll}
P & \in & \mathit{Program} & ::= \quad \overline{D}\ e \\
D & \in & \mathit{Class} & ::= \quad \texttt{class}\ c\ \{\overline{f}, \overline{\mathit{method}}\} \\
\mathit{method} & \in & \mathit{Method} & ::= \quad m(\overline{x})\{e\} \\
l & \in & \mathit{Location} & ::= \quad p.f \\
p & \in & \mathit{ObjAddr} & \\
v & \in & \mathit{Value} & ::= \quad p \mid \texttt{null} \\
e & \in & \mathit{Expression} & ::= \quad \texttt{new}\ c() \mid x \mid v \mid e.f \mid e.f = e \mid e.m(\overline{e}) \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{acq}\ e \\
 & & & \mid \quad \texttt{rel}\ e \mid \texttt{fork}\ e \\
 & & & \\
E & \in & \mathit{Context} & ::= \quad E.f \mid E.f = e \mid p.f = E \mid E.m(\overline{e}) \mid p.m(\overline{v}, E, \overline{e}) \mid \texttt{acq}\ E \mid \texttt{rel}\ E \\
 & & & \mid \quad \texttt{let}\ x = E\ \texttt{in}\ e \\
H & \in & \mathit{Heap} & ::= \quad (\mathit{ObjAddr} \to \mathit{Tid}_{\perp}) \text{ and } (\mathit{Location} \to \mathit{Value}) \\
T & \in & \mathit{ThreadSet} & ::= \quad \mathit{Tid} \to e \\
a, b & \in & \mathit{Action} & ::= \quad t : \texttt{acq}\ p \mid t : \texttt{rel}\ p \mid t : \texttt{read}\ l\ v \mid t : \texttt{write}\ l\ v \mid t : \texttt{fork}\ p\ t' \mid t : \texttt{no-op} \\
\alpha, \beta & \in & \mathit{Trace} & ::= \quad \overline{a}
\end{array}
$$

$$\boxed{P \vdash H, T \to^{a} H, T}$$

$$
\begin{array}{llll}
P \vdash H, T[t := E[p.m(\overline{v})]] & \to^{t:\texttt{no-op}} & H, T[t := E[e[\overline{x} := \overline{v}, \texttt{this} := p]] & \text{if } p \text{ contains } m(\overline{x})\{e\} \\
P \vdash H, T[t := E[\texttt{let}\ x = v\ \texttt{in}\ e]] & \to^{t:\texttt{no-op}} & H, T[t := E[e[x := v]]] & \\
P \vdash H, T[t := E[p.f = v]] & \to^{t:\texttt{write}\ p.f\ v} & H[p.f := v], T[t := E[v]] & \\
P \vdash H, T[t := E[p.f]] & \to^{t:\texttt{read}\ p.f\ v} & H, T[t := E[v]] & H[p.f] = v \\
P \vdash H, T[t := E[\texttt{acq}\ p]] & \to^{t:\texttt{acq}\ p} & H[p := t], T[t := E[\texttt{null}]] & H[p] = \texttt{null} \\
P \vdash H, T[t := E[\texttt{rel}\ p]] & \to^{t:\texttt{rel}\ p} & H[p := \texttt{null}], T[t := E[\texttt{null}]] & \\
P \vdash H, T[t := E[\texttt{fork}\ p]] & \to^{t:\texttt{fork}\ p\ t'} & H, T[t' := p.\texttt{run}(), t := E[\texttt{null}]] & t' \text{ is fresh} \\
P \vdash H, T[t := E[\texttt{new}\ c()]] & \to^{t:\texttt{no-op}} & H, T[t := E[p]] & \text{where } p \text{ is fresh}
\end{array}
$$

Figure 4.3: SimpleJava Syntax and Semantics

performs a single evaluation step

$$P \vdash H, T \to^{a} H', T'$$

and also extends the set G' of global or escaped addresses. The judgement produces two actions a and b. The action b is a `no-op` if the action a is a field access whose race check can be elided; otherwise b is simply the action a of the target program. Thus, combining multiple steps of this judgement yields a run of the race check elision algorithm

$$P \vdash G, H, T \longrightarrow^{\alpha}_{\beta} G', H', T'$$

where α is the full trace of the target program, and the trace β is a subsequence of α that elides accesses to thread-local objects.

Acquire and release actions are never elided because of the unsoundness of partial lock elision, as shown in Section 4.3. The errors involved in partial lock elision for compilers also make it unsound for use in race detection as the race detector may detect a race that does not occur. Reads and writes are elided if the address being

$$\boxed{P \vdash G, H, T \rightarrow_{a}^{a} G, H, T}$$

$$\frac{P \vdash H, T \rightarrow^{t:\texttt{no-op}} H, T'}{P \vdash G, H, T \rightarrow_{t:\texttt{no-op}}^{t:\texttt{no-op}} G, H, T'} \qquad \frac{P \vdash H, T \rightarrow^{t:\texttt{acq}\ p} H, T'}{P \vdash G, H, T \rightarrow_{t:\texttt{acq}\ p}^{t:\texttt{acq}\ p} G, H, T'} \qquad \frac{P \vdash H, T \rightarrow^{t:\texttt{rel}\ p} H, T'}{P \vdash G, H, T \rightarrow_{t:\texttt{rel}\ p}^{t:\texttt{rel}\ p} G, H, T'}$$

$$\frac{\begin{array}{c} p \notin G \\ P \vdash H, T \rightarrow^{t:\texttt{write}\ p.f\ v} H', T' \end{array}}{P \vdash G, H, T \rightarrow_{t:\texttt{no-op}}^{t:\texttt{write}\ p.f\ v} G, H', T'} \qquad \frac{\begin{array}{c} p \in G \\ G' = G \cup \mathit{reachable}(H, v) \\ P \vdash H, T \rightarrow^{t:\texttt{write}\ p.f\ v} H', T' \end{array}}{P \vdash G, H, T \rightarrow_{t:\texttt{write}\ p.f\ v}^{t:\texttt{write}\ p.f\ v} G', H', T'} \qquad \frac{\begin{array}{c} G' = G \cup \mathit{reachable}(H, p) \\ P \vdash H, T \rightarrow^{t:\texttt{fork}\ p\ t'} H, T' \end{array}}{P \vdash G, H, T \rightarrow_{t:\texttt{fork}\ p\ t'}^{t:\texttt{fork}\ p\ t'} G', H, T'}$$

$$\frac{\begin{array}{c} p \notin G \\ P \vdash H, T \rightarrow^{t:\texttt{read}\ p.f\ v} H, T' \end{array}}{P \vdash G, H, T \rightarrow_{t:\texttt{no-op}}^{t:\texttt{read}\ p.f\ v} G, H, T'} \qquad \frac{\begin{array}{c} p \in G \\ P \vdash H, T \rightarrow^{t:\texttt{read}\ p.f\ v} H, T' \end{array}}{P \vdash G, H, T \rightarrow_{t:\texttt{read}\ p.f\ v}^{t:\texttt{read}\ p.f\ v} G, H, T'}$$

Figure 4.4: Dynamic Escape Analysis Algorithm

read/written is not in G (they have not escaped), but are kept if the address is in G (reachable by multiple threads). A write to an object in G expands G to include this new object as well as all objects reachable from it given the current heap. Finally, a fork is never elided for the same reasons as lock acquire and releases and also expands the global set to include all items reachable from the forked thread (as they are also reachable from the *forking* thread).

4.5.3 Partial Race Check Elision is Trace Precise

We prove that if $P \vdash \emptyset, \emptyset, [t := e] \longrightarrow_{\beta}^{\alpha} G, H, T$ then α has a race *if and only if* β has a race.

We start by formalizing the notion of *reachability* with respect to a given heap. We say an address p' is *reachable* from p in heap H if:

- $p' = p$, or
- for some field f, p' is reachable from $H(p.f)$

Moreover, we say an address p is *reachable* from an expression e in a heap H if p is reachable from some addresses q in e in heap H. We use $\mathit{reachable}(e, H)$ to denote the set of addresses reachable from expression e in H.

We say a state G, H, T is *valid* if G contains all references reachable by multiple threads in T with heap H.

Definition 4.5.1. *G, H, T is* valid *if $\forall t_1, t_2 \in Tid$. if $t_1 \neq t_2$ then $reachable(T(t_1), H) \ \cap \ \ reachable(T(t_2), H) \subseteq G$*

The set of addresses an action accesses is defined as:

$$\begin{aligned} addr(t : \texttt{write}\ q.f\ v) &= \{q, v\} \cap ObjAddr. \\ addr(t : \texttt{read}\ q.f\ v) &= \{q, v\} \cap ObjAddr. \end{aligned}$$

Additionally, each access has a target address.

$$\begin{aligned} target(t : \texttt{write}\ q.f\ v) &= q. \\ target(t : \texttt{read}\ q.f\ v) &= q. \end{aligned}$$

Finally, the function *tid* extracts the thread of an action:

$$tid(t : a) = t$$

Next we prove preservation for our algorithm: if the analysis takes a step from a valid state, the resulting state is still valid.

Lemma 12 (Preservation)**.** *If G, H, T is valid and $P \vdash G, H, T \rightarrow^a_b G', H', T'$ then G', H', T' is valid.*

Proof. Case analysis of $P \vdash G, H, T \rightarrow^a_b G', H', T'$ $\square$

All actions in β appear in α. As we do not remove lock acquires, lock releases or forks with our analysis, no happens before information is lost. As such, if two actions are conflicting in β then they are also conflicting in α. Therefore a race in β implies a race in α.

Theorem 4.5.1. *If G, H, T is valid, $P \vdash G, H, T \longrightarrow^\alpha_\beta G', H', T'$, and β has a race then α has a trace.*

Proof. Suppose β has a race between two concurrent conflicting actions b_1 and b_2, then b_1 and b_2 also appear in α. By case analysis on $P \vdash G, H, T \longrightarrow^\alpha_\beta G', H', T'$, no acquire, release, or fork actions are elided so these actions remain the same in both α and β. Therefore b_1 and b_2 are also concurrent and conflicting in α, and so α has a race. □

To prove the other implication we must show that no accesses involved in the first race in α have been elided in β.

Theorem 4.5.2. *If G, H, T is valid, $P \vdash G, H, T \longrightarrow_{\beta}^{\alpha} G', H', T'$, and α has a race then β has a race.*

Proof. Let a_1, a_2 be the first race in α where $p = \mathit{target}(a_1)$ and $p = \mathit{target}(a_2)$. By induction on the length of α, without loss of generality assume $\alpha = a_1.\alpha'.a_2$ where $a_1.\alpha'$ is race-free:

- If $a_1 \in \beta$ then $p \in G$ so $a_2 \in \beta$ (since G is monotonically increasing by Lemma 14 below) so β has a race.
- If $a_1 \notin \beta$ then we have:
  $\alpha = a_1.\alpha'.a_2$
  $p \notin G$
  $p \in \mathit{addr}(a_1)$
  $p \in \mathit{addr}(a_2)$
  α' is race-free
  By lemma 13 below, $a_1 <_{\alpha} a_2$ and we have a contradiction.

□

We next prove two auxiliary lemmas required by the above proof: First, if two actions a_1 and a_2 access the same address p, where p is not in G at the time of a_1, and no race occurs between a_1 and a_2, then a_1 and a_2 are ordered by happens-before. This ordering arises from the race-free transmission of p from $\mathit{tid}(a_1)$ to $\mathit{tid}(a_2)$. There must exist some pair of actions a_1' and a_2' which write p to a shared object and read p from that object in a race-free manner. As a_1' and a_2' are ordered by happens-before, this same ordering applies to a_1 and a_2.

Lemma 13. *Suppose*

$$G, H, T \textit{ is valid}$$

$$P \vdash G, H, T \longrightarrow^{\alpha}_{\beta} G', H', T'$$

$$\alpha = a_1.\alpha'.a_2$$

$$p \notin G$$

$$p \in \mathit{addr}(a_1)$$

$$p \in \mathit{addr}(a_2)$$

$$\alpha' \textit{ is race-free}$$

Then $a_1 <_\alpha a_2$

Proof. By induction on the length of α. Let $t_1 = \mathit{tid}(a_1)$ and $t_2 = \mathit{tid}(a_2)$. We proceed by case analysis on how thread t_2 recieved the address p.

- If $t_1 = t_2$ then $a_1 <_\alpha a_2$ by program order.
- If α contains $t_1 : \texttt{fork}\ p\ t_2$ then $a_1 <_\alpha a_2$ by the fork ordering.
- Otherwise t_2 read p from some shared location $q.f$ previously written by some thread t_3 (which may or may not be t_1). Thus $\alpha' = \alpha_1.a'_1.\alpha_2.a'_2.\alpha_3$ where
  - $a'_1 = t_3 : \texttt{write}\ q.f\ p$ and
  - $a'_2 = t_2 : \texttt{read}\ q.f\ p$

  then:

$p \notin G, p \in addr(a_1)$	Given
$p \in addr(a'_1)$	By Construction
Let $\alpha' = \alpha_1.a'_1.\alpha_2.a'_2.\alpha_3$	
$a_1.\alpha_1$ is race-free	Because α' is race-free
$\lvert\alpha_1\rvert < \lvert\alpha\rvert$	
$a_1 <_\alpha a'_1$	By induction on $\lvert\alpha\rvert$
$a'_1 <_\alpha a'_2$	By α' is race-free
$a'_2 <_\alpha a_2$	By program order
$a_1 <_\alpha a_2$	By transitivity

□

We prove that α has a race if and only if β has a race by using the previous theorems that prove both sides of the implication.

Theorem 4.5.3 (Trace Precision)**.** *If $P \vdash G, H, T \longrightarrow^{\alpha}_{\beta} G', H', T'$ and G, H, T is valid then α has a race if and only if β has a race.*

Proof. By Theorem 4.5.2 and 4.5.1. □

The previous proofs rely on the fact that G is monotonically increasing.

Lemma 14. *If $P \vdash G, H, T \rightarrow^{a}_{b} G', H', T'$ then $G' \supseteq G$.*

Proof. By case analysis on $P \vdash G, H, T \rightarrow^{a}_{b} G', H', T'$. □

4.6 Related Work

A memory model describes what behaviors are permitted by a program, and consequently what optimizations and program transformations a compiler may perform. Sequential consistency [7] is a simple memory model but it prohibits many common and desirable optimizations. The Java Memory Model is a weaker memory model, and therefore enables more optimizations.

Ferrara describes the consequences of the Java memory model for static analysis, including total escape analysis. [43]. Unfortunately, the allowed optimizations under the Java memory model are complex and Sevcik and Aspinall [91] detail a variety of unsound compiler optimizations dealing with race conditions.

Compilers use escape analysis for a variety of optimizations. The two most common being lock elision for thread-local locations and allocating temporary objects on the stack as opposed to the heap. Choi *et al.* provide the current standard for total escape analysis as implemented in the Java Hotspot compiler [27, 78]. They describe a variety of optimizations that can be performed with this analysis and provide a proof of correctness for their total escape analysis and optimizations. They later improve on this work with a faster analysis that does not lose precision [28].

A variety of papers extend this basic escape analysis to either add functionality, improve speed, or improve precision [102, 53, 13]. Salcianu and Rinard [86] use a modified total escape analysis for allocating memory in a region based manner in order to aid garbage collection. Stadler *et al.* extend the total escape analysis computation into a partial escape analysis [96]. The conversion from total to partial escape analysis allows for optimizations of objects that only escape on some program paths. Unfortunately, this added complexity is unsound when applied to lock elision, as shown in Section 4.3. They also make use of inlining to improve their partial escape analysis, a technique also used by Shankar *et al.* [92].

In addition to compilers, many race detection algorithms use escape analyses. Naik *et al.* use a total escape analysis in their static race detector [72], as do Voung *et al.* [100]. Their analysis performs multiple passes, with a final expensive lockset pass at the end to compute a set of locations where races may occur. Their earlier total escape analysis pass removes any variables that do not escape into another thread from the analysis. This race check elision pass does not reduce their precision beyond that of their other optimization passes.

This technique of using a fast, static escape analysis to improve the speed of another, later analysis is also used by Von Praun and Gross [99] and is also used by our own Bigfoot analysis. They add a total escape analysis to the dynamic lockset algorithm Eraser [87]. They use this total escape analysis to perform race check elision for memory that has not escaped into multiple threads. This optimization does not weaken their correctness guarantees.

Nishiyama uses partial escape analysis in a dynamic race detector [74]. They implement a low level lockset-based algorithm in the Hotspot Java virtual machine. Their analysis uses a partial escape analysis based on a read barrier to produce a subset of objects that must be instrumented. Objects are not included in the analysis until the read barrier detects reads from multiple threads on the same address. This analysis does not alter the preciseness of the lockset analysis it is based on but the lockset analysis is already an imprecise analysis.

Likewise, Christiaens and Bosschere make use of a similar partial escape analysis to filter results for a vector clock based dynamic race detector [30]. They implement the vector clock checks at a Java machine code level. They also implement a partial escape analysis at this level that matches our own closely. In addition, they integrate the analysis with the Java garbage collector in order to remove previously global objects from the global set as they become unreachable. Their analysis is already only trace precise so partial lock elision does not reduce their precision. Partial escape analyses can also be implemented at a higher level then the Java machine code as done by Harrington and Freund [58].

4.7 Conclusion

A partial escape analysis provides extra information about when an object escapes compared to a total escape analysis. Unfortunately, using this information for partial lock elision introduces data races into code where none were present before. Additionally, when used in race detectors, partial race check elision reduces the precision of the race detector to trace precise. Partial escape analysis remains useful for race detectors for which trace precision is enough as is the case in this thesis as described in section xxx.

Chapter 5

Capsule-Local Macro Classifications

5.1 Introduction

Our partial thread-local analysis works by tracking what memory is reachable by multiple threads. Through this work we observe that many objects that become reachable by multiple threads share common design patterns of encapsulation and synchronization. For our final analysis we extend our partial thread-local analysis to include these common patterns by introducing the notion of *capsules*. Capsules provide a way of limiting access to their fields (through encapsulation) as well as providing thread safety (through synchronization). This chapter aims to provide a rigorous definition of capsules, provide a proof that race detection checks on capsule-local objects may be elided safely, and provides a classifier that we run on the JavaGrande [59] and DaCapo [12] series of benchmarks to show the widespread usage of capsules in real world code.

In Chapter 4, we explored the validity of eliding checks on partial thread-local objects. In theory, this analysis elides checks on:

- short lived objects that do not escape their allocating frame and
- long lived objects that are never shared between threads.

In practice, we see examples of the first case successfully filtered but few examples of the second. However, the lack of elisions on long lived objects is not because all long lived objects are accessed by multiple threads, but because reachability does not take into account the tools programmers use to encapsulate objects. For example, when a parent thread creates a child thread in Java, all fields of the child thread are reachable by the parent thread. Basing our analysis on a reachability argument means that it is hard to design long lived objects that are not thread-shared unless they appear only on the parent thread. For this analysis we look at *accessibility* instead of just *reachability*. Where reachability does not take into account Java's access modifiers, accessibility does. While a child objects fields are always reachable by a parent thread they are not always accessible.

For objects that are accessed by multiple threads we find that most are synchronized to allow at most one thread access to their fields at any given time. These objects are designed to be shared between threads, but only such that a single thread is *inside* (has a method of that object on its stack) at a time. For example, `Vector` is a class designed to be shared between threads but includes synchronization such that only one thread may be evaluating inside it at any given time. This restriction means that all accesses to a `Vector` object's fields are guaranteed to be race-free (as shown in Section 5.4.1). In addition, `Vector` is written in such a way that the array backing the `Vector` is only accessible from inside the `Vector` class's methods and so will also remain race-free for the length of the program. These timing and encapsulation guarantees combined mean that all accesses to `Vector` and its child objects are race-free.

We call classes that limit outside access to their fields and only allow a single thread to be executing inside them at any one time *capsules*. Specifically, capsules must adhere to these two capsule properties:

- Encapsulation - all field accesses must come from `this`, that is, there is no field access outside of the capsule's method bodies, and

- Synchronization - there is a total happens-before ordering on method calls to the capsule.

When an object marked as a capsule breaks one of these two properties we refer to it as a *capsule violation*.

Note that `Vector` is an extreme case in that it enforces the synchronization property itself. However, self enforcement is not necessary for correctness. For example, `ArrayList` objects also share these two properties, however the responsibility for correctly synchronizing `ArrayList` falls on the library client instead of the library implementation. Despite relying on the implementation to correctly synchronize, the design and documentation of the `ArrayList` class makes it clear that multiple threads concurrently using an `ArrayList`'s methods is an error.

With the change from reachability to accessibility and thread-local to capsule-local many objects that were considered thread-shared in our previous analysis are considered capsule-local in our new analysis. However the change from reachability to accessibility comes with a higher analysis cost. Only writes change the reachability graph while writes, method calls, and method returns all change the accessibility graph. To avoid excess overhead we only consider accessibility as it applies to capsules and conservatively assume that only capsules protect their fields from outside access.

Capsule information can be leveraged to remove unneeded checks on capsule-local objects without reducing the accuracy of race detection. Additionally, our capsule analysis helps to draw out program structure by providing a more accurate picture of what references are accessible from what capsules. Finally, in some cases a capsule violation is an error. For example, a capsule violation on `ArrayList` does not indicate that `ArrayList` should not be treated as a capsule but instead that there was an improper usage of an `ArrayList` object.

To explore these patterns, we define a language on which we prove the correctness of eliminating checks on accesses to capsule-local memory by showing that accesses to capsule-local memory can not be involved in the first race of a program. We translate this formalism to the RoadRunner tool [48] where it acts as a filter for race detection

```
class Parent extends Thread {
    private Object p = new Object();
    public void run(){
        Child c = new Child();
        c.start();
        // c.o is reachable but not accessible here.
    }
}

class Child extends Thread {
    private Object o = new Object();
    public void run(){
        Object m = new Object();
    }
}
```

Figure 5.1: Thread-Local Example

and produces a count of how many locations only access capsule-local objects.

To evaluate the helpfulness of these patterns in analyzing real world code we implement a capsule filtering analysis and a capsule detection analysis in the RoadRunner framework described in Section 2.4. We run these analyses on the same benchmarks as we use for Bigfoot, JavaGrande [59] and DaCapo [12]. We find that a majority of program locations only access capsule-local objects and may be safely filtered by a race detector.

5.2 Capsules

5.2.1 Capsule Requirements

In our previous analysis in Chapter 4, thread reachability formed the basis for our analysis. Any object only reachable by a single thread can only be accessed by that thread and so all accesses to the object are race-free. For example, in figure 5.1 the object at field `p` is only reachable by the `Parent` thread and the object at variable `m` is only reachable by the `Child` thread. Unfortunately, the object at `c.o` is reachable by both threads. In practice we see a large number of fields like `c.o`, that are reachable by multiple threads but, due to Java's typing and the use of access modifiers, are not accessible by multiple threads. By transitioning from reachability to accessibility, we aim to allow standard object-oriented encapsulation techniques to limit the accessibility of objects.

Our definition of reachability from Chapter 4 is:
an address p' is *reachable* from p in heap H if:

- $p' = p$, or
- for some field f, p' is reachable from $H(p.f)$

Here, the main thread can reach `c` and `c` can reach the object at `c.o` and therefore, the main thread can also reach the object at `c.o`.

For our capsule filtering analysis we rely on a limited version of accessibility instead of reachability. Our accessibility analysis is limited in that we conservatively assume that only capsules protect their fields from outside access. While it would be possible to use a full accessibility analysis the added overhead and complexity is not worth it for our race detection purposes. We define accessibility as:
an address p' is *accessible* from p in heap H if p' is not a capsule and either:

- $p' = p$
- q is accessible from $H(p.f)$ for some field f.

If we consider `Child` objects capsules then `c` can access the object at `c.o`, but the `Parent` thread can not access `c` or the object at `c.o`.

We chose to make the capsule distinction at the class level (all objects of a given class are either capsules or none are) in order to limit memory overhead and because we find that this specificity is sufficient in practice.

In the example in Figure 5.1, the object at `c.o` can never become accessible by the `Parent` thread. However, in many cases, private variables may eventually become accessible by multiple capsules through assignment, parameter passing, or return values. For example, in Figure 5.2 the object at `t2.o` may eventually become accessible by `Thread1` either through a call to `t2.getO` or a call to `t2.leak`.

Only assignment modifies reachability while assignment, parameter passing, and return values modify accessibility as seen in `getO` and `leak`. Therefore, to use accessibility we must also instrument capsule method calls and returns to properly track the accessibility graph. Specifically, all objects returned from or passed to capsule methods are marked as shared as they have escaped their containing capsule. Comments in Figure 5.2 show where this sharing occurs.

Our previous analysis is precise because if all accesses come from the same thread, they are guaranteed to be ordered by happens before. While thread locality does guarantee happens before ordering, many objects enforce their own happens before ordering as well. For example, some classes like `Vector` ensure a happens before ordering on their method calls through the use of the `synchronized` keyword. Method call total ordering is easy to instrument as our accessibility analysis already relies on instrumenting method calls and returns for capsule methods. This dynamic approach to monitoring synchronization works for both for classes with internal synchronization, such as `Vector`, but also for classes that rely external synchronizing, such as `ArrayList`.

From these ideas of encapsulation and synchronization we arrive at our two capsule properties seen in the introduction:

- Encapsulation - all field accesses must come from `this`, that is, there is no field access outside of the capsule's method bodies, and
- Synchronization - there is a total happens-before ordering on method calls to the capsule.

With these properties, accesses to capsule-local objects are race-free. Note, capsules themselves are guaranteed to be capsule-local as they can never become accessible by another capsule. The full proof of correctness is shown in Section 5.4.1, but at

```
class Thread1 extends Thread {
    public void run() {
        Thread2 t2 = new Thread2(this);
    }

    public void use(Object o) {
        // o’s fields are accessible.
    }
}

class Thread2 extends Thread {
    private Object o = new Object();
    private Thread1 t1;

    public Thread2(Thread1 t1) {
        this.t1 = t1;
    }

    public void run() {
    }

    public void getO() {
        // Mark o as shared before returning it.
        return o;
    }

    public void leak() {
        // Mark o as shared before passing it as a parameter.
        t1.use(o);
    }
}
```

Figure 5.2: Leaky Capsules

a high level it resembles the thread-local proof of correctness. If all accesses to a particular field come from within a capsule's method bodies (encapsulation), and all method calls are totally ordered (synchronization), then all accesses are also totally ordered. In the case where a reference becomes capsule-shared through assignment, method call, or method return then the race-free sharing of the object ensures that the last capsule-local access and the first capsule-shared access is also race-free.

In practice we loosen the synchronization restriction on capsules slightly. We find that many capsules contain methods that do not access state and are not synchronized. These methods would usually break the capsule's synchronization property but are race-free because they have no accesses. For example, many capsules have unsynchronized helper methods that call synchronized methods. To avoid classifying these helper methods as capsule violations, we lazily enforce the capsule synchronization property on the first memory access of a capsule method instead of immediately upon method entry. This technique also simplifies the edge case of whether or not the lock acquire of a synchronized method happens before or after the method enter action.

The two capsule properties provide a flexible and safe base for classifying classes as capsules. For a non-thread example of a capsule, Figure 5.3 shows the `Counter` class that meets the capsule properties. The `Counter` objects field `a` is never accessed from outside of its method bodies and all three calls to its methods are totally ordered. This ordering of method calls may be enforced by the `Counter` class itself as is shown with the `set` method, or by the caller as is shown with the `unsynchSet` method. Lazy synchronization enforcement allows the helper method `set0` to not cause a capsule violation as it never accesses state.

In this example, `Thread1`, `Thread2`, and `Counter` all meet the requirements for capsules while `int[]` does not. All accesses to the fields of `Thread1`, `Thread2`, and `Counter` are race-free because they are capsule accesses. All accesses to the fields of the array at `c.a` are race-free because the array is only accessible by `c`, and therefore these accesses are capsule-local.

```
class Thread1 extends Thread {
    public void run() {
        Counter c = new Counter();
        Thread2 t2 = new Thread2(c);
        c.set(1, 2);
        c.set0(1)
    }
}

class Thread2 extends Thread {
    private Counter c;

    public Thread2(Counter c) {
        this.c = c;
    }

    public void run() {
        synchronize(c){
            c.unsynchSet(2,3);
        }
    }
}

class Counter {
    private int[] a = new int[10];

    public synchronized set(int i, int v) {
        a[i] = v;
    }

    public unsynchSet(int i, int v) {
        a[i] = v;
    }

    public set0(int v){
        set(0,v);
    }
}
```

Figure 5.3: Objects as Capsules

Our definition of capsules allows for all capsule-local references to have their race checks elided. Looking at our notion of accessibility, we also see that it is impossible for capsules themselves to become capsule-shared. Therefore, direct access to a capsule's fields from within that capsule's methods are always race-free, we refer to these accesses as *capsule* accesses. Access to the fields of non-capsule, capsule-local objects are also race-free but these objects may become capsule-shared in the future. We refer to these accesses as *capsule-protected* accesses.

We provide an extension to EscapeJava (the language used in Chapter 4) called CapJava and a capsule filtering algorithm for CapJava. We provide a proof of correctness for this algorithm showing that filtering capsule-local accesses does not impact the precision of a trace precise race detector. Finally, we implement our filtering algorithm using the RoadRunner framework and run it on the JavaGrande and Da Capo series of benchmarks and find that a majority of locations only access capsule-local objects.

5.2.2 Capsule Detection

In real world applications, people may choose to hand label classes that they intend to be capsules. However, for testing purposes and to aid in ease of use, we have developed a dynamic capsule detection algorithm that can be run alongside capsule filtering and race detection.

The high level idea for capsule detection is to begin by assuming all classes as capsules: as classes make capsule violations, they are removed from the list of capsules. Our tool iteratively executes the program until an execution with no capsule violations occurs. The presence of a capsule violation in an execution may hide a data race during execution, so for the purposes of our algorithm a capsule violation is considered an error. A capsule violation may be the result of a miss-classified capsule or an actual error (such as a capsule violation on ArrayList). If the class is miss-classified the capsule detection algorithm removes it from the list of capsules for the next run. If the capsule violation indicates a real error then the programmer must fix this error. In the benchmarks tested we have found that a single iteration finds at least one capsule violation from each non-capsule class and all future runs are free from capsule violations.

In more detail, capsule detection verifies that all capsule field accesses only come from accesses to `this`, and that all enters and exits of capsule methods are ordered by happens before. To verify that all accesses to capsule fields come from `this` capsule detection inserts a check into race detection's standard access instrumentation. On an access to a field of a capsule, capsule detection compares the class of the target and the location of the access, and if they do not match, the analysis reports a capsule violation. In the future, much of this work could be done with a static analysis by checking that all accesses to fields of capsule objects are of the form `this.x`.

To enforce the capsule synchronization property, that all enter and exits to methods are totally ordered, capsule detection adds instrumentation to the beginning and end of all capsule methods. Each capsule keeps a clock that tracks the timing of its method enters and exits. Upon entering a capsule method, capsule detection compares the clock of the capsule being entered to the clock of the calling thread to ensure that this enter happens strictly after the last exit from this capsule. If the previous exit does not happen before the current enter, the analysis reports a capsule violation. Upon exiting a capsule, capsule detection records the time of the exit to be compared with the threads time at the next enter. Because of the total ordering requirement, capsule detection only needs to track the time and thread of the last exit and not the time of the last exit for each thread.

An execution of a target program running capsule detection, capsule filtering, and race detection produces:

- no errors: this execution is guaranteed to be race-free;
- a capsule violation: this execution may or may not have a race but does have either an incorrectly labeled capsule or an improper usage of a capsule. The programmer either fixes the capsule violation or on future runs the violating class will not be

considered a capsule;

- a data race: this execution is guaranteed to have a data race on the address reported.

In our benchmarks we find the capsule detection algorithm always gives capsule violations on the first execution (as all classes are initially marked as capsules) while all future executions give either no errors or a data race.

5.3 Capsule Theory

We now define a syntax and semantics for CapJava on which we formalize our capsule theories. CapJava aims to mimic Java in a simplified setting that removes all actions not relevant to capsules or race detection.

5.3.1 Syntax

We extend the initial syntax of EscapeJava to include an optional capsule annotation for classes, a new expression for evaluation inside a capsule, and new actions for entering, exiting, reading, and writing to a capsule. We call this extended language CapJava. A program P consists of a sequence of class definitions D (containing methods and fields), as well as a main expression e. Expressions can create new objects (`new`), read from fields ($e.f$), assign to fields ($e.f = e$), create local variables (`let` $x = e$ `in` e), call methods ($e.m(\overline{e})$), acquire and release locks (`acq` e and `rel` e), and fork off new threads (`fork` e).

The [*capsule*] annotation is optionally added to classes that should be treated as capsules. If a capsule annotation is wrongly given, evaluation becomes stuck (necessitating changing the annotation and re-evaluation). In implementation we automatically label classes using capsule detection as described in Section 5.2.2. The capsule annotation is class based, and either all objects of a class are capsules or none are. An object is a capsule, if the class of the object has the capsule annotation.

P	$\in$	$\mathit{Program}$	$::=$	$\overline{D}\ e$
c	$\in$	$\mathit{ClassName}$		
f	$\in$	$\mathit{FieldName}$		
m	$\in$	$\mathit{MethodName}$		
D	$\in$	Class	$::=$	$\texttt{[capsule] class}\ c\ \{\overline{f}, \overline{m}\}$
method	$\in$	Method	$::=$	$m(\overline{x})\{e\}$
l	$\in$	$\mathit{Location}$	$::=$	$p.f$
p, q, r	$\in$	$\mathit{ObjAddr}$		
v	$\in$	Value	$::=$	$p \mid \texttt{null}$
e	$\in$	$\mathit{Expression}$	$::=$	$\texttt{new}\ c() \mid x \mid v \mid e.f \mid e.f = e \mid e.m(\overline{e})$
			$\mid$	$\texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{acq}\ e \mid \texttt{rel}\ e$
			$\mid$	$\texttt{fork}\ e \mid \texttt{in-cap}\ k\ e$
E	$\in$	$\mathit{Context}$	$::=$	$E.f \mid E.f = e \mid p.f = E \mid E.m(\overline{e}) \mid p.m(\overline{v}, E, \overline{e})$
			$\mid$	$\texttt{acq}\ E \mid \texttt{rel}\ E$
			$\mid$	$\texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{in-cap}\ k\ E$
H	$\in$	Heap	$=$	$(p \rightarrow t \mid \bot)$ and $(l \rightarrow v)$
G	$\in$	$\mathit{GlobalSet}$		
T	$\in$	$\mathit{ThreadSet}$	$=$	$t \rightarrow e$
Σ	$\in$	State	$::=$	$H \bullet T$
t	$\in$	Tid	$::=$	(Thread identifiers)
a, b	$\in$	Action	$::=$	$t : \texttt{acq}\ p \mid t : \texttt{rel}\ p \mid t : \texttt{read}\ l\ v \mid t : \texttt{write}\ l\ v$
			$\mid$	$t : \texttt{no-op} \mid t : \texttt{fork}\ p\ t' \mid t : \texttt{enter}\ k\ \overline{v} \mid t : \texttt{exit}\ k\ v$
α, β	$\in$	Trace	$::=$	$\overline{a}$
k	$\in$	$\mathit{Capsule}$	$=$	$\{p : p$ is a capsule$\}$
o	$\in$	Owner	$::=$	$t \mid k$

Figure 5.4: CapJava Syntax

Evaluation uses the `in-cap` k e syntax to track the call stack of capsule methods. The expression `in-cap` k e causes the expression e to be evaluated inside the capsule k. When e has been fully evaluated to a value v, the capsule is exited. The `in-cap` syntax is only ever used by capsule method calls and is not present in user code. The well-foundedness of an expression ensures the correct placement of `in-cap` expressions, as discussed in Section 5.4.1.

Most actions are the same as those in EscapeJava and contain the executing thread, the action, as well as any information needed by the race detector. In addition to these standard actions, we add an enter and exit action to track the flow into and out of capsules. An enter action, `enter` k $\overline{v}$, contains the capsule being entered, k, as well as all of the parameters passed into the capsule by the method call $\overline{v}$. The exit action, t : `exit` k v, contains the capsule being exited, k, as well as the single return value from that method, v. All actions have an owner. If an action is emitted under an `in-cap` k e expression, the owner is the capsule k. Otherwise, the owner of an action is the executing thread. Owners are not used in our semantics but are important for the proof of correctness.

5.3.2 Semantics

[Method]	$P \vdash\ H \bullet T[t := E[p.m(\overline{v})]]$	$\rightarrow^{t:\texttt{no-op}}$	$H \bullet T[t := E[e[\overline{x} := \overline{v}, \texttt{this} := p]]]$	$\texttt{class}(p) = c, c$ contains $m(\overline{x})\{e\}$ and p is not a capsule
[CapMethod]	$P \vdash\ H \bullet T[t := E[k.m(\overline{v})]]$	$\rightarrow^{t:\texttt{enter}\ k\ \overline{v}}$	$H[k := t] \bullet T[t := E[\texttt{in-cap}\ \ k\ e[\overline{x} := \overline{v}, \texttt{this} := k]]]$	$\texttt{class}(k) = c, c$ contains $m(\overline{x})\{e\}$ and $\texttt{capFree}(H, k, a, \alpha)$
[ExitCap]	$P \vdash\ H \bullet T[t := E[\texttt{in-cap}\ \ k\ v]]$	$\rightarrow^{t:\texttt{exit}\ k\ v}$	$H[k := \perp] \bullet T[t := E[v]]$	
[Let]	$P \vdash\ H \bullet T[t := E[\texttt{let}\ x = v\ \texttt{in}\ e]]$	$\rightarrow^{t:\texttt{no-op}}$	$H \bullet T[t := E[e[x := v]]]$	
[Write]	$P \vdash\ H \bullet T[t := E[p.f = v]]$	$\rightarrow^{t:\texttt{write}\ p.f\ v}$	$H[p.f := v] \bullet T[t := E[v]]$	p is not a capsule
[CapWrite]	$P \vdash\ H \bullet T[t := E[k.f = v]]$	$\rightarrow^{t:\texttt{write}\ k.f\ v}$	$H \bullet T[t := E[v]]$	$\texttt{executor}(T(t)) = k$
[Read]	$P \vdash\ H \bullet T[t := E[p.f]]$	$\rightarrow^{t:\texttt{read}\ p.f\ v}$	$H \bullet T[t := E[v]]$	$H(p.f) = v$ and p is not a capsule
[CapRead]	$P \vdash\ H \bullet T[t := E[k.f]]$	$\rightarrow^{t:\texttt{read}\ k.f\ v}$	$H \bullet T[t := E[v]]$	$H(k.f) = v$ and $\texttt{executor}(T(t)) = k$
[Acq]	$P \vdash\ H \bullet T[t := E[\texttt{acq}\ p]]$	$\rightarrow^{t:\texttt{acq}\ p}$	$H[p := t] \bullet T[t := E[\texttt{null}]]$	$H(p) = \texttt{null}$ and p is not a capsule
[Rel]	$P \vdash\ H \bullet T[t := E[\texttt{rel}\ p]]$	$\rightarrow^{t:\texttt{rel}\ p}$	$H[p := \texttt{null}] \bullet T[t := E[\texttt{null}]]$	If $H(p) = t$ and p is not a capsule
[Fork]	$P \vdash\ H \bullet T[t := E[\texttt{fork}\ p]]$	$\rightarrow^{t:\texttt{fork}\ p\ t'}$	$H \bullet T[t' := p.\texttt{run()}, t := E[\texttt{null}]]$	t' is fresh
[New]	$P \vdash\ H \bullet T[t := E[\texttt{new}\ c()]]$	$\rightarrow^{t:\texttt{no-op}}$	$H \bullet T[t := E[p]]$	p is fresh and $\texttt{class}(p) = c$

Figure 5.5: EscapeJava: Semantics

A running program has a heap H and a thread set T. The heap maps locations to values and objects to thread identifiers (Tids). For standard objects, that may act as locks, this Tid tracks the holding thread. For capsules, which are unable to act as locks, the Tid tracks what thread occupies the capsule. Values v are addresses p and `null`. A location $l = p.f$ is an object address p along with a field f. The thread set T maps thread identifiers Tid to expressions e.

A program starts with an empty heap $\emptyset$ and a thread set $T = [t := e]$, with a single thread e and thread identifier t. A single evaluation step

$$P \vdash H, T \rightarrow^{a} H', T'$$

produces an action a. Taken in sequence these actions form a trace α. We include P in the evaluation relation to facilitate method look up, and we assume method names are unique.

The semantics of EscapeJava have been extended to handle operations on capsules separately from those on other objects. Compared with other objects, capsules emit enter and exit actions for capsule method calls, enforce encapsulation and synchronization constraints, and may not act as locks. In implementation, the enter and exit actions are already accommodated by RoadRunner, the privacy constraints are checked dynamically on access, and the synchronization constraints are checked by a small amount of code insertion.

The rules [Method] and [CapMethod] handle method calls. For method calls on capsules, evaluation uses [CapMethod]. This rule emits an t : `enter` $k\ \overline{v}$ action, where k is the capsule whose method is being called, and $\overline{v}$ are the parameters being passed to that method. Executing [CapMethod] adds an `in-cap` $k\ e$ expression to the context where e is the method body being executed by the capsule k. After evaluation of the method body finishes (e has been evaluated to v), evaluation uses the [ExitCap] rule to remove the `in-cap` $k\ v$ expression and exit the capsule. While exiting, evaluation emits an exit action, t : `exit` $k\ v$, where k is the capsule being exited and v is the return value from the current method call.

In order to enforce the proper timing restrictions, evaluation uses the `capFree`(H, k, a, α) syntax. This function checks that the current action (entering the capsule) happens after all previous exit actions from that capsule and checks that no other thread is currently in the capsule.

$$\boxed{\texttt{capFree:} \quad \mathit{Heap} \times \mathit{Capsule} \times \mathit{Action} \times \overline{\mathit{Action}} \rightarrow \mathit{Boolean}}$$

$$\texttt{capFree}(H, k, a, \alpha) = (H(k) = \bot) \text{ and } \forall\ t : \texttt{exit}\ k\ v \in \alpha\ .\ t : \texttt{exit}\ k\ v <_{\alpha} a$$

If the `capFree`(H, k, a, α) check returns true, evaluation enters the capsule. Evaluation records the fact that a thread is currently inside the capsule by modifying $H(k)$ from $\bot$ to the entering thread's Tid. Exiting does not require any checks, but does modify the heap again to return it to its initial state, $H[k := \bot]$

For standard method calls, a $t :$ `no-op` action is emitted and no `in-cap` expression is added. As no `in-cap` expression is added for standard method calls, there is no corresponding exit rule.

The [Let] rule works as expected. The basic reading and writing rules, [Read] and [Write], also work as expected by modifying the store and emitting their corresponding action. These rules do not apply to capsules, as noted in the side conditions. Reading and writing to fields of a capsule use the [CapRead] and [CapWrite] rules. The only difference between the standard and capsule access rules is that the capsule rules enforce encapsulation by ensuring that the executor of the current thread is the capsule being accessed. In other words, fields of a capsule can only be accessed inside of that capsule's methods. To make this restriction [CapWrite] and [CapRead] use the `executor`(t) function. This function returns the capsule that the thread t is currently evaluating under. If the thread is not evaluating under a capsule, the executor of a thread is the thread itself:

$$\boxed{\texttt{executor:} \quad \mathit{Expression} \to \mathit{Owner}}$$

$$\begin{aligned}\texttt{executor}(e) = \;& k \text{ if } e = E[\texttt{in-cap}\ k\ e_2] \text{ and } e_2 \neq E'[\texttt{in-cap}\ \_\ \_] \\ & t \text{ otherwise}\end{aligned}$$

It is possible for evaluation to become stuck on either a faulty capsule method call or access. By becoming stuck, evaluation stops before it breaks one of the capsule properties and so the analysis remains precise. In implementation, instead of becoming stuck, these operations emit a capsule violation and mark the violating class as not a capsule for future runs.

5.3.3 Capsule Filtering

Figure 5.6 shows our capsule filtering algorithm. This algorithm performs a dynamic capsule accessibility analysis, recording in G all addresses accessible by multiple capsules. The judgment

$$P \vdash G, \Sigma \to_b^a G', H'T'$$

performs a single evaluation step

$$P \vdash H, T \to^a H', T'$$

and also extends the set G' of capsule-shared addresses. The judgment produces two actions, a and b. The action b is a `no-op` action, if the action a is an access whose race check can be elided; otherwise b is the same action as a. Thus, combining multiple steps of this judgment yields a run of the capsule filtering algorithm,

$$P \vdash G, \Sigma \longrightarrow_\beta^\alpha G', H'T'$$

$$\boxed{P \vdash G, \Sigma \rightarrow^a_a G, \Sigma}$$

$$\frac{a = t : \texttt{no-op} \quad P \vdash \Sigma \rightarrow^a \Sigma'}{P \vdash G, \Sigma \rightarrow^a_a G, \Sigma'} \text{[CE-Noop]}$$

$$\frac{a = t : \texttt{fork}\ p\ t' \quad P \vdash H \bullet T \rightarrow^a H' \bullet T' \quad G' = G \cup \texttt{accessible}(p, H)}{P \vdash G, H \bullet T \rightarrow^a_a G', H' \bullet T'} \text{[CE-Fork]}$$

$$\frac{a = t : \texttt{acq}\ p \quad P \vdash \Sigma \rightarrow^a \Sigma'}{P \vdash G, \Sigma \rightarrow^a_a G, \Sigma'} \text{[CE-Acq]}$$

$$\frac{a = t : \texttt{rel}\ p \quad P \vdash \Sigma \rightarrow^a \Sigma'}{P \vdash G, \Sigma \rightarrow^a_a G, \Sigma'} \text{[CE-Rel]}$$

$$\frac{a = t : \texttt{write}\ p.f\ v \quad P \vdash \Sigma \rightarrow^a \Sigma' \quad p \notin G}{P \vdash G, \Sigma \rightarrow^a_{t:\texttt{no-op}} G, \Sigma'} \text{[CE-Write1]}$$

$$\frac{a = t : \texttt{write}\ p.f\ v \quad P \vdash H \bullet T \rightarrow^a H' \bullet T' \quad G' = G \cup \texttt{accessible}(v, H) \quad p \in G}{P \vdash G, H \bullet T \rightarrow^a_a G', H' \bullet T'} \text{[CE-Write2]}$$

$$\frac{a = t : \texttt{read}\ p.f\ v \quad P \vdash \Sigma \rightarrow^a \Sigma' \quad p \notin G}{P \vdash G, \Sigma \rightarrow^a_{t:\texttt{no-op}} G, \Sigma'} \text{[CE-Read1]}$$

$$\frac{a = t : \texttt{read}\ p.f\ v \quad P \vdash \Sigma \rightarrow^a \Sigma' \quad p \in G}{P \vdash G, \Sigma \rightarrow^a_a G, \Sigma'} \text{[CE-Read2]}$$

$$\frac{a = t : \texttt{enter}\ k\ \overline{v} \quad P \vdash H \bullet T \rightarrow^a H' \bullet T' \quad G' = G \cup \texttt{accessible}(\overline{v}, H)}{P \vdash G, H \bullet T \rightarrow^a_a G', H' \bullet T'} \text{[CE-Enter]}$$

$$\frac{a = t : \texttt{exit}\ k\ v \quad P \vdash H \bullet T \rightarrow^a H' \bullet T' \quad G' = G \cup \texttt{accessible}(v, H)}{P \vdash G, H \bullet T \rightarrow^a_a G', H' \bullet T'} \text{[CE-Exit]}$$

Figure 5.6: Dynamic Accessibility Analysis and Capsule Filtering Algorithm

in which α is the full trace of the target program, while the trace β contains `no-op` actions for accesses to capsule-local targets.

The rules [CE-Noop], [CE-Acq], and [CE-Rel] do not ever filter their actions or modify the global set G. The rules [CE-Write1] and [CE-Read1] handle filtered accesses and only apply when the object being accessed is capsule-local, $p \notin G$. Accesses matching these rules are race-free so the algorithm produces a `no-op` for the b action. The other access rules, [CE-Write2] and [CE-Read2] handle cases where the object being accessed is capsule-shared, $p \in G$. In these cases, actions are not filtered and the [CE-Write2] rule extends the global set using the $\texttt{accessible}(v, H)$ function shown below.

We say an address q is *accessible* from p in heap H if q is not a capsule and either:

- $p = q$
- q is accessible from $H(p.f)$ for some field f.

This function is similar to the reachability function used in Chapter 4, however capsules and their fields are not considered accessible as direct access to their fields is disallowed by evaluation.

The rules for entering and exiting a capsule, [CE-Enter] and [CE-Exit], are never filtered. These rules produce a new global set G' that is formed from the union of the old global set G and the set of all things accessible by the parameters being passed in, $\overline{v}$ (in a [CE-Enter]), or returned, v (in [CE-Exit]), from the capsule.

5.4 Correctness Proof

5.4.1 Terminology

Our accessibility function takes into account the limited accessibility of capsules. We also modify the definition of free addresses in the same way. Specifically, in the expression $\texttt{in-cap}\ k\ e$, the capsule k protects the expression e so $\texttt{in-cap}$ expressions do not contain free addresses. Also, a capsule is never a free address. All other definitions are standard. We define the free addresses of a term e as:

$$\boxed{\texttt{FA:}\quad \textit{Expression} \to 2^{\textit{ObjAddr}}}$$

$$\begin{array}{lcl}
\texttt{FA}(k) & = & \emptyset \\
\texttt{FA}(p) & = & \{p\} \text{ p is not a capsule} \\
\texttt{FA}(x) & = & \emptyset \\
\texttt{FA}(e.f) & = & \texttt{FA}(e) \\
\texttt{FA}(\texttt{new}\ c()) & = & \emptyset \\
\texttt{FA}(e_1.f = e_2) & = & \texttt{FA}(e_1) \cup \texttt{FA}(e_2) \\
\texttt{FA}(e_1.m(\overline{e_2})) & = & \texttt{FA}(e_1) \cup \texttt{FA}(\overline{e_2}) \\
\texttt{FA}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) & = & \texttt{FA}(e_1) \cup \texttt{FA}(e_2) \\
\texttt{FA}(\texttt{acq}\ e) & = & \texttt{FA}(e) \\
\texttt{FA}(\texttt{rel}\ e) & = & \texttt{FA}(e) \\
\texttt{FA}(\texttt{fork}\ e) & = & \texttt{FA}(e) \\
\texttt{FA}(\texttt{in-cap}\ k\ e) & = & \emptyset
\end{array}$$

We define *roots* for owners (capsules or threads) to take into account both the free addresses in their expressions as well as those accessible from fields. The roots of a thread t with a thread set T are the free addresses in $T(t)$.

$$\boxed{\texttt{roots:}\quad \textit{Owner} \times \textit{ThreadSet} \to 2^{\textit{ObjAddr}}}$$

$$\texttt{roots}(t, T) = \texttt{FA}(T(t))$$

The roots of a capsule k, with respect to a thread set T, are the capsule itself as well as the free addresses of expressions evaluating under that capsule.

$$\texttt{roots}(k, T) = \{k\} \cup \{p \in \texttt{FA}(e) \mid \exists t. T(t) = \texttt{in-cap}\ k\ e\}$$

Finally, p is *root-accessible* from owner o with respect to state $H \bullet T$ if p is accessible from $q \in \texttt{roots}(o, T)$ with respect to heap H.

$$\boxed{\texttt{rootAccessible:} \quad \mathit{Owner} \times \mathit{State} \to 2^{\mathit{ObjAddr}}}$$

$$\texttt{rootAccessible}(o, H \bullet T) = \{p \mid p \in \texttt{accessible}(q, H), q \in \texttt{roots}(o, T)$$

Root-accessible takes into account both the accessibility of objects from other objects, as well as the accessibility of objects from expressions.

A global set and state $G, H \bullet T$ is *valid* if G contains all references root-accessible by multiple owners in a heap H and a thread set T and, for any capsule k, There is at most one $\texttt{in-cap}\ k\ e$ in the thread set T. In other words, at any given point in evaluation, G is the global set or bigger; no two threads are evaluating in the same capsule, and no single thread is evaluating under the same capsule twice.

Definition 5.4.1. *$G, H \bullet T$ is valid if*

$$\begin{gathered}\forall o_1, o_2 \in \mathit{owners}\ \mathit{if}\ o_1 \neq o_2\ \mathit{then} \\ \mathit{rootAccessible}(o_1, H \bullet T)\ \cap\ \mathit{rootAccessible}(o_2, H \bullet T) \subseteq G \\ \mathit{and}\ \forall k \in \mathit{Capsule}\ \mathit{there\ is\ at\ most\ one}\ \texttt{in-cap}\ k\ e\ \mathit{in}\ T \\ \mathit{and}\ H(k) = t\end{gathered}$$

Each access has a target address.

$$\boxed{\texttt{target:} \quad \mathit{Action} \to \mathit{ObjAddr}}$$

$$\begin{aligned}\texttt{target}(t : \texttt{write}\ p.f\ v) &= p \\ \texttt{target}(t : \texttt{read}\ p.f\ v) &= p\end{aligned}$$

The function $\texttt{tid}$ extracts the thread of an action:

$$\boxed{\texttt{tid:} \quad \mathit{Action} \to \mathit{Tid}}$$

$$\texttt{tid}(t : \_) = t$$

A thread t is *in* a capsule k with respect to a threadset T:

$$\boxed{\texttt{in:} \quad \mathit{Tid} \times \mathit{Capsule} \times \mathit{ThreadSet} \rightarrow \mathit{Bool}}$$

$$\text{If there exists } \texttt{in-cap}\ \ k\ e_2 \text{ in } T(t)$$

In Section 5.3.2 we defined the *executor* of a thread as the most recent capsule that thread is in or, if it is in no capsules, the thread itself,

$$\boxed{\texttt{executor:} \quad \mathit{Expression} \rightarrow \mathit{Owner}}$$

$$\begin{aligned} \texttt{executor}(e) = \quad & k \text{ if } e = E[\texttt{in-cap}\ \ k\ e_2] \text{ and } e_2 \neq E'[\texttt{in-cap}\ \ \_\ \_] \\ & t \text{ otherwise} \end{aligned}$$

We also extend this definition to actions:

$$\boxed{\texttt{executor:} \quad \mathit{Action} \times \mathit{ThreadSet} \rightarrow \mathit{Owner}}$$

$$\texttt{executor}(a, T) = \texttt{executor}(T(\texttt{tid}(a)))$$

Evaluating a program P will produce a trace α. Evaluating a program P with our capsule filtering algorithm 5.6 will produce two traces α and β. For every action in trace α at index i, there is a corresponding action in trace β at index i that is either identical to the action in α or a `no-op` . Our proof shows that trace β contains a race if and only if trace α contains a race.

5.4.2 Proof of Correctness

In this section, we prove that the original trace produced by evaluation has a race if and only if the filtered trace also has a race. If the filtered trace contains a race, then it is trivial to prove the original trace also contains the same race, as all accesses in the filtered trace also appear in the original and all acquire and release events are identical. However, if a race occurs in the original, it is not immediately obvious that the same race also occurs in the filtered trace, as one of the offending accesses may have been filtered.

We start by proving that evaluation begins in a valid state and will preserve this valid state throughout execution. We then show that if a race exits between two actions a_1 and a_2, where a_1 has been filtered, then some action shares the target of the race with the owner of a_2. If a_1 and a_2 race then this sharing also races. This sharing race will be caught by race detection, so our analysis remains trace precise even when a_2 is filtered. If this sharing race does not exist, then the original race can not exist either.

To begin, we prove our analysis preserves validity. Given a transition $P \vdash G, \Sigma \longrightarrow^{a_1.a_2...a_i}_{b_1.b_2...b_i} \Sigma', T'$ for an action a_i, we name the prestate of the action Σ_i and the post state Σ'_i. For example, $P \vdash G_1, \Sigma_1 \rightarrow^{a_1}_{b_1} G'_1, \Sigma'_1$. For a state Σ_x we name the heap and thread set $H_x \bullet T_x$.

Lemma 15 (Preservation)**.** *If G, Σ is valid and $P \vdash G, \Sigma \rightarrow^a_b G', \Sigma'$, then G', Σ' is valid.*

Proof. Case analysis of $P \vdash G, \Sigma \rightarrow^a_b G', \Sigma'$ □

The proof of our desired correctness property in one direction is straightforward.

Theorem 5.4.1. *If G, Σ is valid and $P \vdash G, \Sigma \longrightarrow^\alpha_\beta G', \Sigma'$ and β has a race then α has a trace.*

Proof. Suppose β has a race between two concurrent conflicting accesses b_1 and b_2, then b_1 and b_2 also appear in α. By case analysis on $P \vdash G, \Sigma \longrightarrow^\alpha_\beta \Sigma', T'$, no acquire, release, enter, exit, or fork actions are elided so these actions remain the same in both

α and β. Therefore, b_1 and b_2 are also concurrent and conflicting in α, hence α has a race. $\square$

To prove the other implication, we must show that no accesses involved in the first race in α have been elided in β.

Theorem 5.4.2. *If G, H, T is valid and $P \vdash G, \Sigma \longrightarrow^{\alpha}_{\beta} \Sigma', T'$ and α has a race, then β has a race.*

Proof. Let a_1, a_2 be the first race in α, where $p = \mathit{target}(a_1)$ and $p = \mathit{target}(a_2)$. By induction on the length of α, without loss of generality, assume $\alpha = a_1.\alpha'.a_2$ where $a_1.\alpha'$ is race-free:

- If $a_1 \in \beta$ then $p \in G$ so $a_2 \in \beta$ (since G is increasing by Lemma 19 below) so β has a race.
- If $a_1 \notin \beta$ then we have:
  $\alpha = a_1.\alpha'.a_2$
  $p \notin G$
  $o_1 = \texttt{executor}(a_1, T_1)$
  $o_2 = \texttt{executor}(a_2, T_2)$
  $p \in \texttt{roots}(o_1, T_1)$
  $p \in \texttt{roots}(o_2, T_2)$
  α' is race-free
  By lemma 16 below, $a_1 <_\alpha a_2$ and we have a contradiction.

$\square$

We next prove two auxiliary lemmas required by the above proof: First, if two actions a_1 and a_2 access the same address p, where p is not in G at the time of a_1, and no race occurs between a_1 and a_2, then a_1 and a_2 are ordered by happens-before. Intuitively, this ordering arises from the race-free transmission of p from $\texttt{executor}(a_1, T)$ to $\texttt{executor}(a_2, T)$. There must exist some pair of actions a'_1 and a'_2 that transmit p, from the owner of a_1 to the owner of a_2 in a race-free manner (either by through a write, method call, or method return). As a_1' and a_2' are ordered by happens-before, this same ordering applies to a_1 and a_2.

Lemma 16. *Suppose*

$$\begin{aligned}
&G, \Sigma \textit{ is valid}\\
&P \vdash G, \Sigma \longrightarrow^{\alpha}_{\beta} G', \Sigma'\\
&\alpha = a_1.\alpha'.a_2\\
&p \notin G\\
&o_1 = \textit{\texttt{executor}}(a_1, T_1)\\
&o_2 = \textit{\texttt{executor}}(a_2, T_2)\\
&p \in \textit{\texttt{roots}}(o_1, T_1)\\
&p \in \textit{\texttt{roots}}(o_2, T_2)\\
&\alpha' \textit{ is race-free}
\end{aligned}$$

Then $a_1 <_\alpha a_2$

Proof. By induction on the length of α. If $o_1 = o_2$ then by Lemma 17 they are ordered. If $o_1 \neq o_2$ then $p \notin \texttt{roots}(o_2, T_1)$ so at some point p became a root in o_2.

So there exists an action a_3 in α', such that $p \notin \texttt{roots}(o_2, T_3)$ and $p \in \texttt{roots}(o_2, T_3')$. We proceed by case analysis on the type of a_3.

- The cases for [Noop], [Acq], [Rel], [Write] do not modify the roots of o_2.
- [Fork] $a_3 = t_3 : \texttt{fork}\ p\ t_2$, $t_2 = o_2$ then:
  $a_1 <_\alpha a_3$ By induction on $|\alpha|$
  $a_3 <_\alpha a_2$ By fork ordering
  $a_1 <_\alpha a_2$ By transitivity.
- [Exit] $a_3 = t_3 : \texttt{exit}\ k_3\ p$, t_3 is in o_2 with respect to T_3
  then:

$a_1 <_\alpha a_3$ By induction on $|\alpha|$

$a_3 <_\alpha a_2$ By Lemma 17

$a_1 <_\alpha a_2$ By transitivity.

- [Enter] $a_3 = t_3 : \texttt{enter}\ o_2\ \overline{q}$, and $p \in \overline{q}$

  then:

  $a_1 <_\alpha a_3$ By induction on $|\alpha|$

  $a_3 <_\alpha a_2$ By Lemma 18

  $a_1 <_\alpha a_2$ By transitivity

- [Read] $a_3 = t_2 : \texttt{read}\ q.f\ p$, $\texttt{executor}(a_3)T_3 = o_2$

  We proceed by case analysis on if $H(q.f) = p$.

  - $H(q.f) \neq p$

    then:

    $\exists a_4 = t_3 : \texttt{write}\ q.f\ p$

    $a_1 <_\alpha a_4$ By induction on $|\alpha|$

    $a_4 <_\alpha a_3$ By no races in α'

    $a_3 <_\alpha a_2$ By Lemma 17

  - $H(q.f) = p$

    $q \notin G$ and so at some point before q enters G there must be an action a_4 where $q \in \texttt{roots}(o_1, T_4)$:

    $\exists a_4\ .\ \texttt{executor}(a_4)T_4 = o_1, q \in \texttt{roots}(o_1, T_4), q \notin G_4$

    $a_1 <_\alpha a_4$ By Lemma 17

    $a_4 <_\alpha a_3$ By induction

    $a_3 <_\alpha a_2$ By Lemma 17

    $a_1 <_\alpha a_2$ By transitivity

□

If two actions take place in the same owner, they are always ordered.

Lemma 17. *If*

$$G, H, T \text{ is valid}$$
$$P \vdash G, \Sigma \longrightarrow^{\alpha}_{\beta} G', \Sigma'$$
$$\alpha = a_1.\alpha'.a_2$$
$$a_1 \text{ is in } o \text{ with respect to } T_{a_1}$$
$$a_2 \text{ is in } o \text{ with respect to } T_{a_2}$$
$$tid(a_1) = t_1$$
$$tid(a_2) = t_2$$

then $a_1 <_\alpha a_2$

Proof. If o is a thread, then $t_1 = t_2$ and by programs order $a_1 <_\alpha a_2$. If o is a capsule, then both actions may still be in the same thread in which case by program order. If the actions are not in the same thread, then there exists an exit at or after a_1, followed by an enter before a_2 (because no two threads may be in the same capsule at the same time). Without loss of generality, assume the exit comes after a_1.

- $a_{x1} = t1 : \texttt{exit}\ o\ \_$
- $a_{e2} = t2 : \texttt{enter}\ o\ \_$

where $\alpha = a_1.\alpha_1.a_{x1}.\alpha_2.a_{e2}.\alpha_3.a_2$

$a_1 <_\alpha a_{x1}$	By program order
$a_{x1} <_\alpha a_{e2}$	By enter exit order
$a_{e2} <_\alpha a_2$	By program order
$a_1 <_\alpha a_2$	By transitivity

□

If it appears earlier in the trace, the action that enters a capsule happens before an action in that capsule.

Lemma 18. *If*

$$G, H, T \textit{ is valid}$$
$$P \vdash G, \Sigma \longrightarrow^{\alpha}_{\beta} G', Sigma'$$
$$\alpha = a_1.\alpha'.a_2$$
$$a_1 = t_1 : \texttt{enter}\ k\ \_$$
$$a_2 \textit{ is in } k$$
$$tid(a_2) = t_2$$

then $a_1 <_\alpha a_2$

Proof. There must exist some enter that comes before a_2 for it to be in k. If this enter is a_1, then both actions happen in the same thread and are ordered by program order. If this enter is not a_1, then there exists an exit followed by an enter in α'.

- $a_x = t_1 : \texttt{exit}\ k\ \_$
- $a_e = t_2 : \texttt{enter}\ k\ \_$

where:

$a_1 <_\alpha a_x$	By program order
$a_x <_\alpha a_e$	By enter exit locking
$a_e <_\alpha a_2$	By program order
$a_1 <_\alpha a_2$	By transitivity

□

Theorem 5.4.2 relies on the fact that G is monotonically increasing.

Lemma 19. *If* $P \vdash G, \Sigma \rightarrow^a_b G', \Sigma'$ *then* $G \subseteq G'$.

Proof. By case analysis on $P \vdash G, \Sigma \rightarrow^a_b G', \Sigma'$. □

Our main correctness result is then a straightforward combination of the above two theorems.

Theorem 5.4.3 (Trace Precision)**.** *If* G, H, T *is valid and*

$$P \vdash G, \Sigma \longrightarrow^{\alpha}_{\beta} G', \Sigma'$$

then α has a race if and only if β has a race.

Proof. By Theorem 5.4.2 and Theorem 5.4.1. □

5.5 Implementation

5.5.1 Capsule Filter

We implement our capsule filter and capsule detection algorithms using the RoadRunner framework detailed in Section 2.4. This framework instruments Java bytecode, with the appropriate hooks to insert our own analysis code at various event sites analogous to the actions of our semantics. We implement custom hooks where our analyses differ from standard race detection on accesses and entering and exiting from capsules. We also define helper functions to handle transitively sharing objects and checking synchronization guarantees on capsules.

Our analyses can be run either on top of vector clock based race detection or on their own. In both cases, capsule detection checks for capsule violations and reports an error if a capsule violation is detected. If capsule filtering and detection are being run stand-alone, they collect information on how many accesses and locations are capsule or capsule-protected, and can be filtered. If they are being run on top of race detection, then accesses that are not filtered are passed on to the underlying race detection algorithm.

In order to track state on a per object or field basis race detectors use shadow state. This shadow state is added to every object or field by the instrumenter and can be used to store analysis data. Initial vector clock based race detectors stored two vector clocks (one read and one write) per field per object. Each of these vector clocks has one entry per thread in the target program. FastTrack [47] later reduced this requirement, in most cases, from a read/write vector clock to a read/write epoch, although in some cases a full vector clock is still required for reads. Our analysis has different shadow state requirements for capsules and standard objects. For standard objects, one flag per object is added that marks the object as either capsule-shared or capsule-local. This flag is in addition to any other shadow state that the underlying race detector may use. This information can alternatively be captured using a global weak set that more closely matches the global set G of our semantics.

A capsule object has a simplified shadow state that replaces the race detector's shadow state. Instead of keeping shadow state for each field, a capsule's shadow state consists of a single epoch for the object that records the timing information of the capsule's last exit. This reduced state saves memory in situations in which a large amount of capsules are present.

Our idealized algorithm uses a number of helper functions not native to Java. The `accessible`(o, H) function computes a set of objects accessible from the address o and heap H. Java does not natively support any ideas about accessibility. This behavior is instead captured in a `leak` function.

`leak` only operates on objects and arrays as it is not possible to pass other types by reference in Java and so they can not become shared. The `if` on line 3 checks if the object to leak is `null`, already accessible by multiple capsules (`isShared`), a boxed primitive type (such as Integer), or itself a capsule. In all of these cases, leaking has no effect and `leak` returns immediately. If the object to leak does not fall into any of these categories, `leak` marks it as capsule-shared and then searches it for more references, recursively leaking any it finds. If the object to leak is an array, then `leak` searches using standard iteration (line 9). If the object to leak is not an array, `leak` uses reflection to access all the fields of the object (line 15). For optimization purposes, code rewriting may be used to add a `getAllFields` method to each object if reflection is too slow for the target use case.

The `leak` function is used by all actions that cause objects to become capsule-shared. An object may become shared through writes, capsule enters, or capsule exits. As capsules classes are known at compile time, calls to leak are placed at the beginning of each capsule method (to leak the parameters) and at the return site of capsule methods

```
void leak(Object o) {
    String oClass = o.getClass();
    if(o == null || isShared(o) || isPrimitive(oClass) || isCAP(oClass))
        return;

    markAsShared(o);

    if (o.getClass().isArray()) {
        // Transitive search if o is an array.
        for(int i=0; i<Array.getLength(o); i++){
            Object child = Array.get(o, i);
            leak(child);
        }
    } else {
        // Transitive search if o is an object.
        Field[] fields = new Field[1];
        fields = getAllModelFields(o.getClass()).toArray(fields);
        for(int i = 0;i<fields.length;i++) {
            if(fields[i] == null) continue;
            fields[i].setAccessible(true);
            leak(fields[i].get(o));
        }
    }
}
```

Figure 5.7: Java `leak` Implementation

(to leak the return value). These instrumentations are only made for reference types. For example, a method that takes an integer as a parameter and returns an integer is unmodified.

Access events check if the access should be filtered, leak objects on writes, and mark capsule violations if the access is to a capsule outside a capsule method. `filterAccess`, shown in Figure 5.8, performs each of these three tasks. RoadRunner calls this function on each access with the `AccessEvent` for the current access. If `filterAccess` returns `true`, then further race detection is skipped without any loss of precision. If `filterAccess` returns `false` the underlying race detector performs its checks as normal.

`filterAccess` begins by identifying the target of the access, `target`, and the value being assigned, `newValue`. `filterAccess` next handles the special case of static fields, something not covered in our semantics. If the access is a static write, `newValue` is immediately leaked. All static fields are accessible by all capsules, so our analysis leaks any value that is assigned to one. All static accesses must be checked so `filterAccess` returns `false`.

`filterAccess` next checks if `target` is a capsule. If `target` is a capsule `filterAccess` checks for a capsule violation by comparing the current program location and `target`'s class (line 20). If this check fails, `target`'s class is added to `capsuleViolations` and will no longer be counted as a capsule for future runs. If `target` is a capsule and does not have a capsule violation, then `filterAccess` filters this access by returning true.

If the target is not a capsule, it may still be capsule-protected (line 26). If the target is capsule-protected, `filterAccess` returns `true`. If the target is neither a capsule nor capsule-protected, `filterAccess` can not filter this access and returns `false` (line 30). In the case where the current access is a write, it also leaks `newValue` as this object is now capsule-shared.

In addition to the code shown, `filterAccess` contains bookkeeping code that is used to classify the various accesses. At the end of a run, this data is used to form the results seen in section 5.6. Some optimizations are present in the actual code compared with the example code shown in this paper. For example, class comparisons are handled using integers instead of string comparison, and some information is stored directly in the shadow state.

5.5.2 Capsule Detection

In order to enforce the capsule properties, capsule detection checks for encapsulation violations at accesses to capsule fields and synchronization violations upon entering capsule methods. As seen previously, capsule detection checks the encapsulation property in `filterAccess` alongside filtering. Capsule detection enforces the synchronization property upon entering a capsule method body. Upon entering a cap-

```
boolean filterAccess(AccessEvent event) {
    Object target = event.getTarget();
    Object newValue = null;
    if(event.newValue.getType() == TaggedValue.Type.OBJECT) {
        newValue = event.newValue.getObjectValue();
    }

    // Leak static writes.
    if(isStatic(event)) {
        if(event.isWrite()) {
            leak(newValue);
        }
        return false;
    }

    String targetClass = target.getClass();
    if(isCAP(targetClass)) {
        // Mark a violation if accessed outside of capsule bounds
        String currentClass = getProgramLocation();
        if(currentClass.equals(targetClass)){
            return true;
        } else {  // Report a capsule violation.
            capsuleViolations.add(targetClass);
            return false;
        }
    } else if(!isShared(target)) {
        return true;
    } else {
        if(event.isWrite()) leak(newValue);
        return false;
    }
}
```

Figure 5.8: Java `filter` Implentation

sule method, `enterCap` (shown in Figure 5.9) checks that there is not another thread already in the capsule (line 3) and that the last exit happens before the current enter (line 8). Capsules use an epoch to track both of these properties. Upon exiting a capsule, the time of the exit is recorded in the capsule epoch. An exit action never fails. Upon entering a capsule, the capsules epoch is checked against the current threads vector clock to ensure the proper happens before ordering. If an enter succeeds, capsule

```
boolean enterCap(Epoch capsuleTimer, int tid) {
    synchronized(capsuleTimer) {
        if(occupied(capsuleTimer)) {
            return false; // A thread is already in this capsule.
        } else {
            VectorClock v = ts_get_V(tid);
            long threadTime = v.get(mockThread);
            if(time(capsuleTimer) <= threadTime) {
                return true; // The last exit happens before this enter.
            } else {
                // The last exit doesn't happen before this enter.
                return false;
            }
        }
    }
}

void exitCap(Epoch capsuleTimer, int tid) {
    synchronized(capsuleTimer) {
        VectorClock v = ts_get_V(thread);
        setTime(capsuleTimer, tid, v.get(tid));
    }
}
```

Figure 5.9: Java Capsule Timing Implentation

detection sets the capsule's epoch's Tid to the current thread and its time component to `-1`. Trying to enter a capsule whose time component is `-1` always fails due to the `occupied` check.

We make one modification to `enterCap` based on common code patterns we find in real world code. Instead of placing `enterCap` immediately at the start of a capsule method call, we lazily place it before the first access within that capsule's method body. This placement allows for helper methods, which may not be synchronized and that do not modify state to not trigger a capsule violation. We see this pattern especially in unsynchronized helper methods that call synchronized methods. For example, in `Vector` the `contains` method is unsynchronized, however it does not access any state on its own and only calls the `indexOf` method that is synchronized.

5.6 Results

	All			CAP			Cap-Protected			Single-Capsule-Access			Read Shared			Other			Mixed		
Benchmark	Arr	Obj	Cla	Arr	Obj	Cla	Arr	Obj	Cla	Arr	Obj	Cla	Arr	Obj	Cla	Arr	Obj	Cla	Arr	Obj	Cla
Avrora	1,110	1,798	161	0	0.51	0.83	0.95	0.01	0.02	0.03	0.36	0.02	0.01	0.02	0	0	0.01	0	0.01	0.09	0.14
Batik	1,345	4,286	321	0	0.54	0.56	0.96	0.16	0.02	0.04	0.28	0.02	0	0	0	0	0	0	0	0.02	0.4
fop	4,522	6,033	322	0	0.39	0.48	0.96	0.09	0.02	0.04	0.48	0.5	0	0	0	0	0	0	0	0.04	0
lufact	78	186	9	0	0.56	0.16	0.54	0	0.26	0	0.06	0.37	0.06	0.01	0.11	0.06	0.21	0.11	0.33	0.16	0
lunindex	413	2,483	118	0	0.45	0.45	0.85	0.18	0.07	0.03	0.28	0.03	0	0.01	0.07	0	0	0.04	0.12	0.08	0.35
lusearch	399	1,305	78	0	0.4	0.17	0.85	0.08	0.31	0.05	0.23	0.45	0	0	0.05	0	0	0.01	0.1	0.29	0
pmd	2,172	4,270	295	0	0.25	0.32	0.87	0.08	0.01	0.1	0.55	0.01	0	0	0	0	0	0	0.03	0.12	0.66
xalan	582	2,407	189	0	0.36	0.11	0.67	0.09	0.29	0.17	0.5	0.28	0.01	0.05	0.15	0	0	0.17	0.15	0	0
AVERAGE				0	0.43	0.39	0.83	0.09	0.13	0.06	0.34	0.21	0.01	0.011	0.05	0.01	0.03	0.04	0.09	0.1	0.19
Crypt	93	144	4	0	0.19	0.5	0.7	0.58	0.25	0.09	0.03	0	0.11	0	0	0	0	0	0.11	0.21	0.25
Elevator	56	188	7	0	0.84	0.86	0.5	0.01	0	0	0	0	0.5	0	0	0	0	0	0	0	0.14
moldyn	147	706	5	0	0.27	0.6	0.1	0.02	0.2	0.31	0.57	0	0.1	0	0	0.07	0	0	0.42	0.15	0.2
montecarlo	96	177	11	0	0.15	0.09	0.32	0.67	0.27	0.03	0.12	0	0.01	0	0	0	0	0	0.64	0.06	0.64
raytracer	16	306	10	0	0.04	0.2	0.69	0.13	0	0.13	0.23	0	0	0	0	0	0	0	0.19	0.61	0.8
sor	31	87	3	0	0.63	1	0.48	0.11	0	0.39	0	0	0.1	0	0	0.03	0	0	0	0.25	0
sparsematmult	85	105	3	0	0.98	1	0.98	0	0	0.02	0	0	0	0	0	0	0	0	0	0.02	0
AVERAGE				0	0.44	0.61	0.54	0.22	0.10	0.14	0.14	0	0.12	0	0	0.014	0	0	0.19	0.19	0.29

Table 5.1: Location categorization for arrays (Arr), objects (Obj), and classes (Cla) as a fraction of the total locations

In addition to our proof of correctness, we also show that checks on capsule and capsule-protected locations make up a majority of checks in Java code. We follow RoadRunner's standard treatment of libraries: fields of Java's core library classes are not checked for races, and synchronization operations internal to those libraries are assumed not to be used to protect any of the target's data and are ignored. However, several key library methods from `java.lang.Object` and `java.lang.Thread`, such as `Object.notify` and `Thread.start`, are treated specially as synchronizing operations. In general, we believe that including all libraries would lead to an increase in capsule and capsule-protected locations, as many of the Java library containers are capsules and the arrays backing Java's collections are almost always capsule-protected.

We measure results for objects, arrays, and classes. Arrays can not be capsules as their contents are always accessed from outside their own method bodies, but they can be capsule-protected. Objects may be either capsules or capsule-protected but not both. We mark objects as capsules based on their violation or not of the capsule properties, regardless of programmer intent. Many helper objects can be considered either capsule-protected or capsules because their parents' synchronization ensures their own synchronization. We classify these as capsules as we have no way of knowing programmer intent. For classes, the capsule classification is straightforward as either all instances of a class are capsules or none are. For capsule-protected classes, we only count those in which all objects of the given class are capsule-protected. All classes that fall into multiple categories are marked as mixed. No library classes are counted towards the total number of classes or any of the subcategories.

Table 5.1 shows the classification of program locations for the JavaGrande (top) and DaCapo (bottom) benchmarks. Our tool classifies locations into one of the following categories:

- Capsule - The target is always a capsule,
- Capsule-Protected - the target is always capsule-local but is not a capsule itself,

- Single-Capsule-Access - the target is only accessed by a single capsule, but is capsule-shared,
- Other - the target is capsule-shared and is accessed by multiple capsules,
- Read Shared - the target is capsule-shared but only written once,
- Mixed - The target is capsule-local on some accesses and capsule-shared on other accesses.

Of these categories, locations that fall in the capsule or capsule-protected categories are always race-free. Those that fall in the single-capsule-access category are race-free at the moment but, as they are accessible by multiple capsules, may eventually be involved in a race. These locations are not safe to filter. Code refactoring or a more complex capsule analysis may be able to convert some of these locations to capsule or capsule-protected locations. Other locations have targets that are capsule-shared and actively accessed by multiple capsules. The targets are not capsules themselves due to either encapsulation or synchronization violations and they are not local to any one capsule. For example, locations that access a shared global array fall into this category. Mixed locations are those that access objects that are capsule-local on some accesses but capsule-shared on others. These locations generally access objects that start as capsule-protected but eventually become capsule-shared.

We consider locations that only access capsule objects or capsule-protected objects filterable. Both the JavaGrande and Da Capo benchmarks show that a majority of program locations can be filtered using the capsule technique with 52% of object and 83% of array sites filterable in JavaGrande and 66% of object and 54% of array sites filterable in Da Capo. Benchmarks show a wide range of variation in capsule results based on their programming patterns from a low of 4% capsules in raytracer to a high of 84% capsules in elevator. Even in programs with few total capsules, such as raytracer, the number of capsule-protected locations remains high (69% for arrays and 14% for objects). The JavaGrande benchmarks show much more uniformity in programming

	Filtered			Unfiltered		
Benchmark	Array	Object	Classes	Array	Object	Classes
Avrora	0.95	0.52	0.85	0.05	0.48	0.16
Batik	0.96	0.7	0.58	0.04	0.3	0.42
fop	0.96	0.48	0.5	0.04	0.52	0.5
lufact	0.54	0.56	0.42	0.45	0.44	0.59
lunindex	0.85	0.63	0.52	0.15	0.37	0.49
lusearch	0.85	0.48	0.48	0.15	0.52	0.51
pmd	0.87	0.33	0.33	0.13	0.67	0.67
xalan	0.67	0.45	0.4	0.33	0.55	0.6
AVERAGE	0.83	0.52	0.51	0.17	0.48	0.49
Crypt	0.7	0.77	0.75	0.31	0.24	0.25
Elevator	0.5	0.85	0.86	0.5	0	0.14
moldyn	0.1	0.29	0.8	0.9	0.72	0.2
montecarlo	0.32	0.82	0.36	0.68	0.18	0.64
raytracer	0.69	0.17	0.2	0.32	0.84	0.8
sor	0.48	0.74	1	0.52	0.25	0
sparsematmult	0.98	0.98	1	0.02	0.02	0
AVERAGE	0.54	0.66	0.71	0.46	0.32	0.29

Table 5.2: Filtered vs. Non-Filtered Locations

style with a high of 54% capsules for Batik and a low of 25% for pmd. Almost all array locations in JavaGrande programs are capsule-protected with an average of 85%. Lufact, the JavaGrande benchmark with the lowest capsule-protected array count, has two large thread-shared arrays that make up the bulk of non capsule-protected array locations in that program.

Table 5.2 shows the percentage of filtered versus unfiltered checks. We consider capsule and capsule-protected locations to be filtered and all others to be unfiltered. As can be seen, the majority of both array and object locations from both benchmark sets can be filtered.

5.7 Discussion

Despite a majority of locations being filtered, there are still objects only accessed by a single capsule that can not be filtered as they are accessible by multiple capsules. These objects are analogous to the objects in our previous analysis that were only ever accessed by a single thread despite being reachable by multiple threads. Locations that access these objects make up the single-capsule-access column. It is not safe to filter these accesses as a future access may eventually come from another capsule. Some locations fall into this category due to the coarseness of our accessibility procedure that can not distinguish between a reference being transferred versus shared. For instance, `Vector`'s `toArray` method produces a fresh array that is not accessible by `Vector` after it is returned. Because the array passes through a capsule return it will always leak regardless of the fact that `Vector` keeps no reference to it and can no longer access it. A more complex analysis, that distinguishes between sharing and transferring accessibility, may be able to mark these objects or arrays as capsule-protected.

We also find a number of instances of classes that qualify as capsules based on their synchronization, but have outside access to some of their fields. Changing these public fields to be private with getter and setter methods would allow these classes to be considered capsules. While currently all capsule-local locations are dynamically filtered, there is the potential to make some of these statically filterable as well. Capsule accesses can often be fully determined with typing information and lend themselves well to static optimization. Some capsule-protected locations can be proven not to leak using an escape analysis but generally this categorization lends itself well to a dynamic filter.

Read shared and single-capsule-access locations can not be filtered safely. Although all current accesses have all been from the same capsule, there is always the possibility for a future access from a different capsule that may race with a previous access. Likewise for read shared there is always the possibility of a future write. For this reason, read shared and single-capsule-access locations' vector clocks must be kept around for this potential future.

5.8 Related Work

This work shares many of the same related works with those detailed in Chapter 4. This work differs in its use of accessibility, specifically its special treatment of capsules.

Many lockset based algorithms use ownership systems, similar to our accessibility system, in order to minimize false positives on thread-local objects. Lockset based algorithms track which locks are held when accessing fields and report an error if the set of locks protecting a field drops to zero. A thread-local object that accesses fields with no locks held can result in a false positive if some type of thread-local versus thread-shared distinction is not made. One of the original lockset papers, Eraser: A Dynamic Data Race Detector for Multithreaded Programs [87], classifies objects as either exclusive to a thread or shared. Instead of using reachability or accessibility based arguments for classification, it uses access patterns. This distinction can cause Eraser to miss races on the first access to an object that does not come from its original thread. As the lockset algorithm is not precise, this drawback matters less as it does not change the total correctness of the algorithm. As vector clock algorithms are precise, we would be significantly lowering the preciseness of race detection if we relied on access instead of accessibility to determine locality.

Hybrid Dynamic Data Race Detection [75] expands on Eraser's approach and uses a vector clock based analysis to fill in some of the shortcomings of the lockset algorithm. They develop a distinct notion of ownership that uses some static analysis to expand on Eraser's notions of thread-local versus thread-shared.

Object Race Detection [99] is another lockset based algorithm that uses a more advanced ownership notion to increase performance. They introduce an analysis that attempts to capture ownership transfer. They note that almost all instances of ownership transfer involve a parent thread that: initializes some objects and a child thread, passes ownership of the objects to the child thread, and never accesses the objects again. Traditional lockset would classify these objects as thread-shared, as they are accessed by both parent and child thread, when really the ownership has been transferred. Object Race Detection still relies on accesses instead of accessibility, which can cause missed races if a race occurs at a transition point. As they have added additional transition points to account for transfer of ownership, these transition points present more areas for false negatives.

Goldilocks: a Race and Transaction-Aware Java Runtime [40] uses the standard lockset based ownership notion but advances the idea to allow container classes to also own references. This idea is similar to our idea of capsules, in that not just thread ownership can protect references but synchronized objects can as well. Beyond this similarity, the algorithm shares more with previous lockset based algorithms than our own, in that it is access based and is specifically meant to work with lockset.

Efficient and Precise Data Race Detection for Multithreaded Object-Oriented Programs [29] is another lockset based algorithm with its own take on ownership. They focus on a large amount of static code analysis in order to improve the accuracy and speed of their lockset based dynamic algorithm. One of these static passes uses escape analysis to mark objects as thread-local. While their approach does not use accesses as the basis of thread ownership like the analysis seen above, it still has trouble with the transition from thread-local to thread-shared and can miss races at these transition points.

TRaDe [30] introduces a novel idea to solve the problems of ownership transfer dealt with by Object Race Detection. Instead of adding more categories to transition between thread-local and thread-shared, TRaDe integrates with the Java garbage collector to analyze when objects become unreachable by a given thread. The garbage collector already does much of this analysis to determine which objects are reachable by zero threads and can have their memory freed. They add a small amount of analysis to this step to also compute which objects are reachable by only a single thread. This extra instrumentation allows them to track objects that start out thread-local, become thread-shared, but then later move back to being thread-local when the original thread loses its reference to the object.

In addition to the dynamic and static analysis described here, there is also work on using types to aid in ownership tracking. For example, the work done in Ownership Types for Safe Programming [21] and in Rust [69]. Both of these systems rely on programmer annotations and unique type systems to statically determine which references are thread-local and therefore race-free. Rust also contains unique syntax to transfer a reference instead of sharing it and to define objects that can be safely thread-shared (similar to our notion of capsules but statically instead of dynamically enforced).

Finally, a variety of tools exist which are not race detection based but use similar ideas to extract program architecture, specifically involving ownership and encapsulation. Dynamic Architecture Extraction [46] uses a dynamic analysis to construct class diagrams about reference reachability, but does not make any claims about race detection utility. Static Architecture Extraction using Annotations [3] has similar goals of categorizing classes based on what classes they can reach or encapsulate, but uses a static analysis instead of a dynamic one. To overcome the limitations of performing such an analysis statically, they rely on programmer annotations to aid their analysis.

Our work brings together many of these ideas but differs mainly in the fact that it is accessibility based and not access based and so remains precise. Additionally, our notion of capsules allows for objects besides threads to protect references from data races and our dynamic capsule detection algorithm allows for capsules to be defined without programmer annotations.

5.9 Conclusion

Our thread-local analysis in Chapter 4 fails to deal with long lived objects, as threads have no way of protecting their references from leaking. Capsules provide a method to protect references, both for threads and other objects that meet the criteria. We show that capsule-local locations are free from data races by using an idealized language and algorithm along with a proof of correctness for that algorithm. We also show that a majority of location targets in popular Java benchmarks are capsule-local and that capsule tracking can be captured using existing race detection tools. The fact that capsules are both safe and already in use by programmers makes them a good candidate for race check elision.

Chapter 6

Summary of Results

Writing multithreaded programs is notoriously difficult in part due to a lack of determinism, atomicity, and sequential consistency. These issues become simplified in the absence of data races and so this thesis focuses on detecting and eliminating data races. We chose to focus on precise dynamic race detectors as they give the precision needed without restricting the realm of valid programs that can be written. These race detectors work well and successfully identify data races without false positives or false negatives. Unfortunately, they are often slow due to the need to check every heap access for a data race. Therefore, the main goal of this thesis is to reduce the overhead of dynamic race detectors by removing unnecessary checks while remaining precise.

While previous work has reduced the overhead of each individual check [47], the sheer number of checks still causes slowdown. When implemented in RoadRunner, the initial vector clock algorithms have a slow down of 20x , the FastTrack optimization brings this down to 8.5x. We reduce this overhead by eliminating checks that do not impact the precision of vector clock algorithms.

We approach removing these checks in two major ways. First, in BigFoot, we look at access patterns mostly inside of a particular critical region and reason about the overlap of checks in these regions. This proves particularly useful in cases where large numbers of checks can be coalesced into a single check or where multiple access to the same locations means that some checks can be removed entirely. This analysis reduces the run-time overhead of dynamic race detection to `2.5x` and also reduces memory overheads as well.

Our second approach deals not with synchronization primitives but with the reachability of objects by multiple threads. An object that can only be reached by a single thread is trivially race-free and therefore does not need to be monitored despite belonging to heap memory. Compilers already use a similar technique to allocate some objects on the stack or elide locking operations for thread-local locks. We begin with a proof of correctness for these use cases and find, surprisingly, that partial lock elision is unsound for compilers and race detectors. We also find that partial race check elision reduces the accuracy of dynamic race detection from address precise to trace precise.

When considering thread-local objects we identify two main categories. The first is temporary objects that are never assigned to fields while the second is objects protected from outside access by encapsulation techniques. Our thread-local analysis correctly classifies most temporary objects as thread-local but does not take into account ideas of encapsulation. Under this analysis if a child thread is reachable by a parent thread then any field of the child is reachable by the parent thread. In order to classify these objects as safe, we transition from reachability to accessibility and add the idea of capsules.

We identify two key properties that allow an object to protect its fields from races, first all accesses to the objects fields must come from `this`, and second all method calls to the object are totally ordered. We call objects that meet both of these properties capsules. Many data structures in the standard Java library fall into this category. We develop a capsule-local analysis that extends our thread-local analysis and allows us to better classify objects which are reachable by multiple threads but are only accessible through a single capsule.

We provide a proof that that all accesses to capsule-local objects are race-free. We implement our capsule filtering algorithm using the RoadRunner framework to classify objects as capsule-local or capsule-shared. We find that a majority of race checks are on locations that only access capsule-local objects and are guaranteed race-free.

These two broad techniques of micro and macro memory classification allow for fewer race checks in a dynamic race detector. With these techniques we introduce no false positives and only lower the preciseness slightly by going from address precise.

Many programming language tools have started with high overheads and over time become more efficient and integrated with the runtime. For example, array bounds checking in memory safe languages initially requires a check at every array access but through static and dynamic techniques is able to eliminate a large number of these checks [105]. It is our hope that similarly race detection will eventually be fast enough to be integrated into the runtime and allow a races-as-exceptions model in order to ease the difficulties in writing multithreaded code.

Bibliography

[1] Louisville-based company's real-time power grid data shows how blackout 'hop scotched' unpredictably and unexpectedly throughout northeast and midwest. http://www.prnewswire.com/news-releases/what-caused-the-power-blackout-to-spread-so-widely-and-so-fast-genscapes-unique-data-will-help-answer-that-question-70952022.html. Accessed: 2016-9-9.

[2] Martín Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *TOPLAS*, 28(2):207–255, 2006.

[3] Marwan Abi-Antoun and Jonathan Aldrich. Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. In *OOPSLA 09*, pages 321–340. ACM, 2009.

[4] Sarita Adve. Data races are evil with no exceptions: technical perspective. *Communications of the ACM*, 53(11):84–84, 2010.

[5] Sarita V Adve and Hans-J Boehm. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, 2010.

[6] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. Technical report, RICE UNIV HOUSTON TX DEPT OF ELECTRICAL AND COMPUTER ENGINEERING, 1995.

[7] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[8] Rahul Agarwal and Scott D. Stoller. Type inference for parameterized race-free Java. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation*, pages 149–160, 2004.

[9] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2006.

[10] Alexander Aiken and David Gay. Barrier inference. In *POPL*, pages 243–354, 1998.

[11] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. Valor: efficient, software-only region conflict exceptions. In *OOPSLA*, pages 241–259, 2015.

[12] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA*, pages 169–190. ACM, 2006.

[13] Bruno Blanchet. Escape analysis for object-oriented languages: Application to java. In *OOPSLA*, pages 20–34, 1999.

[14] Jayaram Bobba, Kevin E Moore, Haris Volos, Luke Yen, Mark D Hill, Michael M Swift, and David A Wood. Performance pathologies in hardware transactional memory. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 81–91. ACM, 2007.

[15] Robert Bocchino, Vikram Adve, Sarita Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 4–4, 2009.

[16] Robert L Bocchino Jr, Vikram S Adve, Danny Dig, Sarita V Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *ACM Sigplan Notices*, volume 44, pages 97–116. ACM, 2009.

[17] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. pages 241–250, 2011.

[18] Hans-J Boehm and Sarita V Adve. Foundations of the c++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008.

[19] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: Proportional detection of data races. In *PLDI*, 2010.

[20] Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. OCTET: capturing and controlling cross-thread dependences efficiently. In *OOPSLA*, pages 693–712, 2013.

[21] Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

[22] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *OOPSLA*, pages 56–69, 2001.

[23] Cardelli, L. A semantics of multiple inheritance. In *Semantics of Data Types*, Lecture Notes in Computer Science 173, Berlin, 1984. Springer Verlag.

[24] Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM, 2007.

[25] A. T. Chamillard, Lori A. Clarke, and George S. Avrunin. An empirical comparison of static concurrency analysis techniques. Technical Report 96-084, Department of Computer Science, University of Massachusetts at Amherst, 1996.

[26] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *International Conference on Functional Programming (ICFP)*, pages 66–77, 2005.

[27] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for Java. In *OOPSLA*, pages 1–19, 1999.

[28] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, 2003.

[29] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, pages 258–269, 2002.

[30] Mark Christiaens and Koenraad De Bosschere. Accordion clocks: Logical clocks for data race detection. In *Euro-Par*, pages 494–503, 2001.

[31] M. Creeger. Multicore cpus for the masses. 2005.

[32] Chris J Date and Hugh Darwen. *A Guide To Sql Standard*, volume 3. Addison-Wesley Reading, 1997.

[33] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[34] Steve Dever, Steve Goldman, and Kenneth Russell. New compiler optimizations in the Java HotSpot™ virtual machine. In *JavaOne Conference*, 2006.

[35] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multiprocessing. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 85–96. ACM, 2009.

[36] DRD: a thread error detector, 2014.

[37] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. Technical Report 94-045, Department of Computer Science, University of Massachusetts at Amherst, 1994.

[38] Laura Effinger-Dean, Hans-Juergen Boehm, Dhruva R. Chakrabarti, and Pramod G. Joisha. Extended sequential reasoning for data-race-free programs. In *MSPC*, pages 22–29, 2011.

[39] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-Juergen Boehm. IFRit: interference-free regions for dynamic data-race detection. In *OOPSLA*, pages 467–484, 2012.

[40] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *PLDI*, pages 245–255, 2007.

[41] Dawson R. Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[42] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *OSDI*, pages 151–162, 2010.

[43] Pietro Ferrara. Static analysis via abstract interpretation of the happens-before memory model. In *TAP*, pages 116–133, 2008.

[44] Cormac Flanagan and Stephen N Freund. Type-based race detection for java. In *PLDI*, volume 35, pages 219–232. ACM, 2000.

[45] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *POPL*, 39(1):256–267, 2004.

[46] Cormac Flanagan and Stephen N Freund. Dynamic architecture extraction. In *Formal Approaches to Software Testing and Runtime Verification*, pages 209–224. Springer, 2006.

[47] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[48] Cormac Flanagan and Stephen N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *PASTE*, pages 1–8, 2010.

[49] Cormac Flanagan and Stephen N. Freund. RedCard: Redundant check elimination for dynamic race detectors. In *ECOOP*, pages 255–280, 2013.

[50] Cormac Flanagan, Stephen N Freund, and Jaeheon Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. *ACM SIGPLAN Notices*, 43(6):293–303, 2008.

[51] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.

[52] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247, 1993.

[53] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Compiler Construction, 9th International Conference*, pages 82–93, 2000.

[54] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA. *TOPLAS*, 17(1):85–122, 1995.

[55] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckle. *The Java Language Specification, Java SE 8 Edition.* 2015.

[56] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.

[57] Dan Grossman. Type-safe multithreading in Cyclone. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*, pages 13–25, 2003.

[58] Emma Harrington and Stephen N Freund. Using escape analysis in dynamic data race detection. *Williams College Technical Report CSTR 201401*, 2014.

[59] Java Grande Forum. Java Grande benchmark suite. Available from `http://www.javagrande.org/`, 2003.

[60] Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *OOPSLA 2009*, pages 97–116, 2009.

[61] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[62] Edward A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[63] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[64] Tongping Liu, Charlie Curtsinger, and Emery D Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 327–336. ACM, 2011.

[65] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A Popa, and Yuanyuan Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 103–116. ACM, 2007.

[66] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, volume 40, pages 37–48. ACM, 2006.

[67] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *ISCA*. ACM, June 2010.

[68] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL*, pages 378–391, 2005.

[69] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[70] Friedemann Mattern. Virtual time and global states of distributed systems. In *Workshop on Parallel and Distributed Algorithms*, 1988.

[71] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[72] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI*, pages 308–319, 2006.

[73] Robert HB Netzer and Barton P Miller. What are race conditions?: Some issues and formalizations. *LOPLAS*, 1(1):74–88, 1992.

[74] Hiroyasu Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Virtual Machine Research and Technology Symposium*, pages 127–138, 2004.

[75] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPOPP*, pages 167–178, 2003.

[76] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. *ACM Sigplan Notices*, 44(3):97–108, 2009.

[77] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I August. Automatic thread extraction with decoupled software pipelining. In *MICRO-38*, pages 12–pp. IEEE, 2005.

[78] Michael Paleczny, Christopher A. Vick, and Cliff Click. The java hotspot server compiler. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*, 2001.

[79] Eli Pozniansky and Assaf Schuster. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[80] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[81] William Pugh. Fixing the java memory model. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 89–98. ACM, 1999.

[82] Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. BigFoot: Static check placement for dynamic race detection. Technical Report CSTR-201702, Williams College, 2017. Available at `http://www.cs.williams.edu/~freund/papers/bigfoot-tr.pdf`.

[83] Dustin Rhodes, Cormac Flanagan, and Stephen N Freund. Correctness of partial escape analysis for multithreading optimization. In *FTFJP*, page 9. ACM, 2017.

[84] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, 1993.

[85] Michiel Ronsse and Koenraad De Bosschere. RecPlay: A fully integrated practical record/replay system. *TCS*, 17(2):133–152, 1999.

[86] Alexandru Salcianu and Martin C. Rinard. Pointer and escape analysis for multithreaded programs. In *PPOPP*, pages 12–23, 2001.

[87] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[88] Edith Schonberg. On-the-fly detection of access anomalies. In *PLDI*, pages 285–297, 1989.

[89] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71, 2009.

[90] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with LLVM compiler - compile-time instrumentation for ThreadSanitizer. In *RV*, pages 110–114, 2011.

[91] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the java memory model. In *European Conference on Object-Oriented Programming*, pages 27–51. Springer, 2008.

[92] Ajeet Shankar, Matthew Arnold, and Rastislav Bodík. Jolt: lightweight dynamic analysis and removal of object churn. In *OOPSLA*, pages 127–142, 2008.

[93] Nir Shavit and Dan Touitou. Software transactional memory. In *SPDC*, pages 204–213, 1995.

[94] Young Wn Song and Yann-Hang Lee. Efficient data race detection for C/C++ programs using dynamic granularity. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 679–688, 2014.

[95] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 682–696. ACM, 2016.

[96] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 165. ACM, 2014.

[97] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, page 13, 1999.

[98] Christoph von Praun and Thomas Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI*, pages 115–128, 2003.

[99] Christoph von Praun and Thomas R. Gross. Object race detection. In *OOPSLA*, pages 70–82, 2001.

[100] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *FSE*, pages 205–214, 2007.

[101] T.J. Watson Libraries for Analysis (WALA). Available at `http://wala.sourceforge.net/`, 2012.

[102] John Whaley and Martin C. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA*, pages 187–206, 1999.

[103] James R. Wilcox, Parker Finch, Cormac Flanagan, and Stephen N. Freund. Array shadow state compression for precise dynamic race detection. In *ASE*, pages 155–165, 2015.

[104] Michael Wolfe. Beyond induction variables. In *PLDI*, pages 162–174, 1992.

[105] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination for the java hotspot&trade; client compiler. In *PPPJ*, pages 125–133, 2007.

[106] Xinwei Xie and Jingling Xue. Acculock: Accurate and efficient detection of data races. In *CGO*, pages 201–212, 2011.

[107] Eran Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL*, pages 27–40, 2001.

[108] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, pages 221–234, 2005.