Dynamic Datarace Detection for Object-Oriented Programs

by

Manu Sridharan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

Author . . . .
Department of Electrical Engineering and Computer Science
May 15, 2002

Certified by . . . .
Jong-Deok Choi
VI-A Company Thesis Supervisor

Certified by . . . .
Martin C. Rinard
M.I.T. Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Dynamic Datarace Detection for Object-Oriented Programs

by

Manu Sridharan

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2002, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

Multithreaded shared-memory programs are susceptible to dataraces, bugs that may exhibit themselves only in rare circumstances and can have detrimental effects on program behavior. Dataraces are often difficult to debug because they are difficult to reproduce and can affect program behavior in subtle ways, so tools which aid in detecting and preventing dataraces can be invaluable. Past dynamic datarace detection tools either incurred large overhead, ranging from 3x to 30x, or sacrificed precision in reducing overhead, reporting many false errors. This thesis presents a novel approach to efficient *and* precise datarace detection for multithreaded object-oriented programs. Our runtime datarace detector incurs an overhead ranging from 13% to 42% for our test suite, well below the overheads reported in previous work. Furthermore, our precise approach reveals dangerous dataraces in real programs with few spurious warnings.

VI-A Company Thesis Supervisor: Jong-Deok Choi
Title: Research Staff Member, IBM T.J. Watson Research Center

M.I.T. Thesis Supervisor: Martin C. Rinard
Title: Associate Professor

Acknowledgments

This thesis presents work done jointly with Jong-Deok Choi, Robert O'Callahan, and Vivek Sarkar of IBM T.J. Watson Research Center, Keunwoo Lee of the University of Washington, and Alexey Loginov of the Universiy of Wisconsin. Some figures appearing in the thesis also appear in [10] and [9]. Many of the programs we used as benchmarks where provided by Christoph von Praun and Thomas Gross of ETH Zurich, to whom we owe great thanks. This work could not have been completed without the Jikes Research Virtual Machine framework, so many thanks to all the researchers in the runtime and optimization groups at Watson who designed and implemented it. Special thanks to Julian Dolby, Steve Fink, Dave Grove, Igor Pechtchanski, Steve Smith, and Martin Trapp for always taking time to patiently answer any question I had about the system.

I feel privileged to have worked under Jong-Deok Choi for two summers and a fall at IBM. Jong taught me a great deal about doing research and life as a researcher, and he gave me an important and intellectually stimulating role in his project, an opportunity available to few with my experience. I'm deeply grateful to Jong for being an incredible supervisor and making my time at IBM fun and exciting.

I'm also thankful for having the chance to get to know Rob O'Callahan and Keunwoo Lee during my last assignment at Watson. I greatly enjoyed working closely with Rob during the fall and trying to absorb a bit of his immense wisdom and knowledge through many fascinating conversations. Keunwoo's passions for research and the arts mirror my own, and I feel incredibly lucky that we were paired on the same research project and are now good friends.

Martin Rinard kindly agreed to supervise this thesis, for which I'm very grateful. Mark Christiaens and Daniel Jackson gave useful feedback on this work which made its presentation clearer and more thorough.

The biggest thanks goes to my parents, whose constant encouragement, support, and love have always inspired and amazed me. Without their strong and guiding influence, I would have never succeeded in completing this work. My brother Vishnu always astounds me with his brilliance, discipline, and kindness, and his support and friendship have also helped me immeasurably.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many important classes of software are implemented using multiple threads of execution and shared memory. Multithreaded programs often have non-deterministic behavior that inhibits reasoning about their properties. An interesting characteristic observed in a particular execution of a non-deterministic program may not be easily reproducible through re-execution. One particularly irksome type of non-deterministic behavior is a *datarace*, roughly defined as when two threads access common memory with no explicit ordering between the accesses, and at least one of the accesses writes the memory. Dataraces almost always indicate programming errors[1], and they can be extremely difficult to debug since their effect on the functional behavior of a program can be indirect and subtle. Furthermore, some dataraces may have no discernable effect for almost all executions of a program, making extensive and repeated testing an ineffective technique for finding them.

Because of the difficulties involved in manually finding and debugging dataraces, a great deal of work has been done to develop tools and techniques which can automatically detect dataraces and/or prevent them from occurring. One class of previous research suggested purely *static* techniques for handling dataraces, addressing the datarace problem for all possible executions of a given program. Some past work

[1]A datarace may not be a programming error if the accessed memory location does not need to hold an exact value for program correctness. Apart benign dataraces on statistics variables like those in our examples (see Chapter 6), chaotic relaxation algorithms can maintain some convergence properties even in the presence of dataraces.

created tools which perform static datarace detection, identifying a set of program statements which in some execution could be involved in a datarace [26, 14, 10]. Unfortunately, because of scalability and precision issues in static analyses such as alias analysis, it is often difficult to statically prove that any possible execution of a program statement will not be involved in a datarace. Therefore, previous static datarace detection tools tend to conservatively flag a large number of safe instructions as dangerous, hindering their usefulness. A more recent static approach involves new programming language constructs and type systems that allow for static verification that a program is datarace-free [1, 5, 4]. While these solutions could influence future language design, commonly used languages such as Java have not incorporated these constructs, and therefore most current software is still datarace-prone.

Another class of past work performed datarace detection *dynamically*, finding dataraces in a particular execution of a given program. Although dynamic datarace detection typically does not prove that a program is free of dataraces since it only utilizes information from one execution, in practice the approach has been shown to be useful for finding bugs. Some previous research has resulted in tools which can fairly precisely identify dataraces in program executions, but these tools added an undesirable large execution time overhead ranging from 3× to 30× [21, 25, 24, 6, 16, 17, 12]. Recent work dramatically reduced this overhead to between 16% and 129% by detecting dataraces at a coarser granularity than individual memory locations [23]. However, the reduced precision of this technique resulted in the reporting of many events which were dataraces according to their coarser definition, but were not actually unsynchronized accesses to shared memory, making their tool less useful for finding real programming errors.

This thesis presents a novel approach to dynamic datarace detection for multi-threaded Java programs. The approach is precise, in that almost all dataraces identified are in fact unsynchronized accesses to shared memory. The approach is also far more efficient than other approaches with similar precision, with time overhead ranging from 13% to 42% for our test cases. These results were achieved through a combination of complementary static and dynamic optimization techniques.

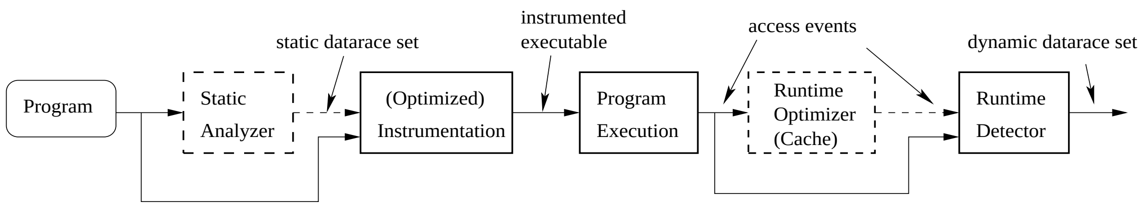


Figure 1-1: Architecture of Datarace Detection System

A high-level view of our datarace detection approach is seen in Figure 1-1 (optional phases are indicated by dashed boxes). The static analyzer phase identifies a *static datarace set*, a conservative set of statements whose executions could potentially be involved in a datarace. If available, this static datarace set is utilized to avoid instrumentation of statements not in the set, as their executions are guaranteed to not be involved in dataraces.

The instrumentation phase inserts tracing calls into the program to monitor executions of statements which could be involved in dataraces. Optionally, an optimization pass can be performed in this phase which removes *redundant instrumentation*, instrumentation which will always produce redundant information (redundancy will be defined more precisely in Chapter 4). The result of this phase is an instrumented executable which will run in the program execution phase.

While running the executable, the instrumentation generates a sequence of *access events* containing sufficient information for performing datarace detection. If employed, the runtime optimizer phase filters these events by caching them and only sending events to the runtime detector that are not redundant compared to a cached event. Finally, given a stream of access events as input, the runtime detector phase performs datarace detection and reports any dataraces to the user.

Since the properties of our datarace detection algorithm enable many of our optimizations, this thesis will present our approach in roughly the reverse order in which it is actually executed. Chapter 2 defines and illustrates datarace terminology in greater detail. Chapter 3 describes our datarace detection algorithm. Chapter 4 presents the dynamic optimizations utilized in our approach, and Chapter 5 delineates our static optimizations. Results are detailed in Chapter 6, followed by a discussion of related work in Chapter 7. Finally, Chapter 8 presents our conclusions and describes future work.

Chapter 2

Datarace Terminology

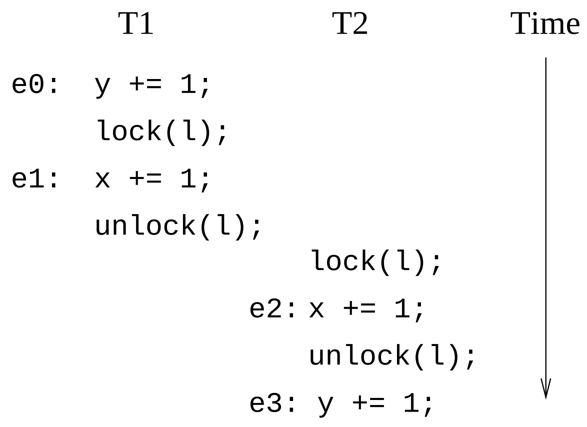
This chapter defines dataraces and related terminology more carefully and illustrates them with examples.

2.1 Datarace Definitions

A number of definitions of dataraces have been described in past work on datarace detection, and the differences between them can be subtle. This section explains some common datarace definitions, including the monitor-based definition utilized in our approach, and clarifies the relationships between them.

2.1.1 Happened-Before Dataraces

The formal definition of a datarace which has the closest correspondence to the typical high-level definition, unordered accesses to a memory location performed by multiple threads, is based on the *happened-before relation*, first defined in [18]. The happened-before relation, written as $\rightarrow$, is an irreflexive partial ordering on all events executed by all threads in an execution of a program. If events e_i and e_j are executed by the same thread and e_i occurs before e_j, then $e_i \rightarrow e_j$. Also, if e_i and e_j are executed by different threads, then $e_i \rightarrow e_j$ if some inter-thread communication construct forces e_i to occur before e_j. For example, in the execution depicted in Figure 2-1, $e1 \rightarrow e2$



```
e0:  y += 1;
     lock(l);
e1:  x += 1;
     unlock(l);
                    lock(l);
               e2: x += 1;
                    unlock(l);
               e3: y += 1;
```

Figure 2-1: Example to illustrate the happened-before relation.

because of the communication between threads T1 and T2 through the locking and unlocking of monitor l. In the Java programming language, the key inter-thread communication constructs are monitors (`synchronized` blocks) and the `start` and `join` methods of the `Thread` class[1]. If e_i and e_j are unordered by the happened-before relation, they are considered *concurrent*, written as $e_i \not\rightarrow e_j$.

Given the happened-before relation, a *happened-before datarace* can be defined as follows. For a memory access event e_i, let $e_i.m$ represent the memory location accessed and $e_i.a$ represent the type of the access (either READ or WRITE). The memory access event pair (e_i, e_j) is a happened-before datarace iff (1) $e_i \neq e_j$, (2) $e_i \not\rightarrow e_j$, (3) $e_i.m = e_j.m$, and (4) $e_i.a = \text{WRITE} \vee e_j.a = \text{WRITE}$. We will discuss this definition in more detail when comparing it to those based solely on monitors.

2.1.2 Monitor-Based Dataraces

For languages such as Java where the primary method of inter-thread synchronization is the use of monitors, alternate definitions of dataraces have been proposed which are based on the use of monitors to protect shared data. Datarace detectors based on these definitions check that threads consistently acquire certain monitors before

[1]The `wait` and `notify` methods of the `Object` class are not discussed separately since their ordering relationships are captured in the locking required to invoke the methods. Also, the current draft of the revised Java Memory Model would force separate consideration of writes and reads of `volatile` fields [20].

accessing shared memory, thereby "locking" the memory since other threads must acquire the same monitors before accessing it. Accordingly, monitors are often called *locks*, and we shall refer to them in this manner throughout the rest of this thesis. We will also use the term *lockset*, introduced in [24], to refer to the set of locks held by a thread at some point in time.

The first definition of dataraces based on locksets was presented in [13], which instead of lockset used the term "lock cover." The original definition was based on both locks and the happened-before relation, but here we present a version solely based on locks. Define an *access event* e as a 5-tuple (m, t, L, a, s) where

- m is the identity of the *logical memory location* being accessed,
- t is the identity of the *thread* which performs the access,
- L is the lockset held by t at the time of the access,
- a is the access type (one of { WRITE, READ }) and
- s is the source location of the access instruction.

The exact definition of a logical memory location can vary, but unless stated otherwise assume that it corresponds to the finest granularity at which reads and writes can occur, for example object fields and array elements in Java. Also note that source location information is only present for error reporting purposes and has no bearing on other definitions and optimizations. In this definition, access events (or, simply, accesses) e_i and e_j are a datarace (written $IsRace(e_i, e_j)$) iff (1) $e_i.m = e_j.m$, (2) $e_i.t \neq e_j.t$, (3) $e_i.L \cap e_j.L = \emptyset$, and (4) $e_i.a = \text{WRITE} \lor e_j.a = \text{WRITE}$. Since this definition checks that some common lock is held in condition (3), but not any specific lock, we call this type of datarace a *common lock datarace.*

The researchers who developed Eraser [24] defined dataraces slightly differently, stating that a datarace occurs when a thread accesses a shared memory location without holding some *unique* lock associated with that location. For example, if three writes to memory location m by three different threads occurred with locksets $\{l_1, l_2\}$, $\{l_2, l_3\}$, and $\{l_1, l_3\}$, Eraser would report a datarace, since no unique lock is

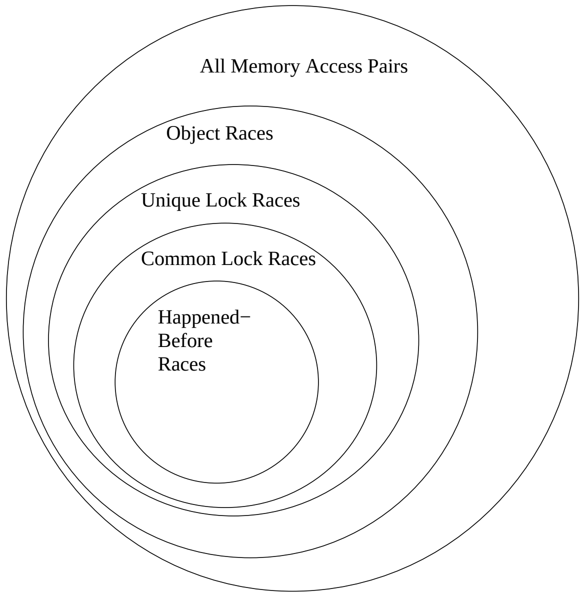


Figure 2-2: Relationship Between Different Datarace Definitions

held during all three accesses, while a detector based on the common lock datarace definition would not report a problem, since all pairs of accesses have some lock in common. We call dataraces conforming to Eraser's definition *unique lock dataraces.*

Recent work defined another type of lockset-based datarace for object-oriented programs, an *object race* [23]. An object race is similar to a unique lock datarace, but its logical memory location is an object rather than the finer granularity locations utilized in the aformentioned definitions. This implies, for example, that two threads accessing *different* fields of an object without holding a common lock constitutes an object race. The justification given for this coarser granularity in the presenting work is that an object is often an abstraction of related data, and therefore it is reasonable to define the object as the unit of protection rather than individual fields [23].

2.1.3 Comparison

Figure 2-2 illustrates the subset relationships of the different datarace definitions for a given execution of a program. Note that this figure is *not* to scale. For our test suite, we found that the happened-before datarace set, the common lock datarace set, and the unique lock datarace set were almost equal in size, while the object race set was significantly larger. The differences between the different lockset-based datarace definitions will be discussed in more detail in Chapter 6, where some statistics will illustrate the distinctions more clearly.

The happened-before datarace definition is appealing because it precisely captures the inter-thread communication that underlies all synchronization idioms, so once the primitive ordering constructs of a language are identified, all higher-level synchronization operations will automatically be handled correctly, a property which does not necessarily hold for lockset-based definitions. For example, consider a typical producer-consumer architecture in which a shared buffer is utilized to send data from the producer to the consumer. If the reads and writes to the shared buffer are properly synchronized, mutation of the data read from and written to the buffer can be safely performed without extra synchronization. A tool which detects happened-before dataraces will recognize this behavior as safe, but a tool which performs lockset-based detection will flag the accesses to the shared data as dataraces, since no locks are held during these accesses. In this sense, the happened-before datarace definition is more precise than any of the definitions based on locks.

However, when considering its usefulness for a datarace detection tool, one key drawback of the happened-before datarace definition is its dependence on thread scheduling, which sometimes masks the existence of synchronization bugs. For example, $e0 \rightarrow e3$ in Figure 2-1 because of the locking and unlocking of l, but in an execution where T2 locks l before T1, $e0 \not\rightarrow e3$, and $(e0, e3)$ would be a happened-before datarace. In comparison, a datarace detection tool based on locks would report a datarace for the execution depicted in Figure 2-1, since no locks are held during the writes to y. Although no dynamic datarace detection tool could indicate all possible dataraces in a program while operating on a single execution, in some sense the happened-before datarace definition is more susceptible than lockset-based definitions to missing bugs because of the quirks of a particular execution. Using the terminology defined in [22], lockset-based detectors detect both *apparent* and *feasible* dataraces, while detectors based on happened-before only detect *apparent* dataraces.

Another important consideration when deciding which datarace definition to use for a detector is impact on efficiency. Previous work has asserted that lockset-based datarace definitions are inherently more amenable to efficient detector implementations than the happened-before datarace definition [24, 23]. This assertion is based on the fact that a detector using the happened-before datarace definition must maintain information about program execution orderings, which potentially scales poorly in space and time as the length of the execution and degree of parallelism increases. A detector for lockset-based dataraces only needs to maintain and compare lockset, thread, and access type information for access events, for the most part ignoring program ordering constraints. However, these constraints on what information each detector must maintain do not necessarily imply that one method will always be more efficient than the other. In fact, recent work has employed several clever optimizations to implement a happened-before based detector for Java which has less overhead than Eraser [12]. While many of the optimizations we employ to reduce the overhead of datarace detection based on locksets do not directly apply for a detector based on happened-before, future research may discover how to adapt our optimizations for happened-before based detectors or may find entirely new optimizations based on happened-before.

2.1.4 Our Approach

Our datarace detection algorithm finds common lock dataraces, as defined previously (the implications of this choice will be discussed in more detail in Chapter 6). We also handle the orderings imposed by the `start` and `join` methods of the `Thread` class using an *ownership model* and *pseudolocks.*

Handling the execution orderings created by `Thread.start()` is important because if they were ignored, some techniques which rely on the ordering would cause many false dataraces to be reported. For example, a thread often initializes some data and then creates a child thread which reads the data with no locking, a safe operation that pure lockset-based detection would flag as a datarace. We use an *ownership model* similar to the one utilized by Eraser [24] to improve our accuracy in these cases. The first thread which accesses a location becomes its owner, and tracking of reads and writes to a location only begins after a thread other than its owner accesses it. While this technique could affect completeness (since the first two accesses to a location by different threads may indeed be a datarace), in practice we have not observed this problem, and the method is effective in suppressing datarace reports for the initialization technique described above. Note that the definition of unique-lock dataraces does not capture our use of the ownership model. We will discuss the implications of the ownership model in more detail in section 5.3, after the rest of our detection system has been presented.

The orderings imposed by `Thread.join()` are also often exploited by programmers to avoid locking. After a `join` call on a child thread returns, the parent thread can safely read data modified by the child thread without synchronization. We utilize dummy synchronization objects called *pseudolocks* to handle these orderings within our lockset-based detection framework. Each thread T_j performs a `monitorenter` on a pseudolock S_j when it begins its execution and a `monitorexit` on S_j when it completes its execution. Any thread which invokes the `join` method of T_j performs a `monitorenter` on S_j when the invocation returns. So, any thread T_i which calls `join` on another thread T_j will hold the pseudolock S_j after thread T_j completes, and T_j holds S_j for the entirety of its execution. Pseudolocks allow us to precisely model the execution ordering introduced by `Thread.join()` between a parent and child thread[2], in some cases leading to far fewer false positive datarace reports.

2.2 Dataraces Reported

Another important characteristic of any datarace detection technique is the set of dataraces it guarantees to report. Given the definition of access events and $IsRace$ given in 2.1.2, we formally define *datarace detection* as follows. Let E be the sequence of access events generated in an execution of a given program. Performing datarace detection on that execution is equivalent to determining the truth value of

[2]Note that this scheme does not precisely model all orderings implied by `Thread.join()`. To be completely precise, when `join()` is invoked on a child thread T_j, the parent thread would have to acquire the pseudolock S_j *and* all the pseudolocks T_j held at its completion.

the following condition:

$$\exists e_i, e_j \in E \mid IsRace(e_i, e_j).$$

Most previous datarace detection systems guaranteed the reporting of all access event pairs in a given execution which were dataraces. However, given an execution with N accesses, the worst-case time and space complexity required for maintaining that guarantee is $O(N^2)$, since all event pairs could be a datarace. Furthermore, [11] shows that a single errant write can lead to many datarace pairs for a given execution, and it is not clear that reporting all of these dataraces would be useful to a programmer. To avoid necessarily high execution costs and possibly exorbitant output, we do not guarantee reporting of all datarace pairs for a given execution. Instead, we report a useful subset of the dataraces which allows for many new optimizations.

Let $MemRace(m_k)$ be the set of access event pairs that are a datarace on memory location m_k. Our datarace detection algorithm guarantees that for memory locations m_k such that $MemRace(m_k)$ is non-empty, we report at least one access event e in $MemRace(m_k)$. For a number of reasons, this seemingly limited information is actually sufficient for determining the cause of most dataraces. e is always reported immediately after it occurs, allowing for suspension of the program's execution to investigate its state more closely. We also guarantee to report the locks held during the access which races with e, and quite often we can report the thread which executed the other access. Our static datarace analyzer, discussed in more detail in section 5.1, usually provides a small, conservative set of source locations whose executions could potentially race with e, giving further aid in finding the source of the datarace. Finally, a record / replay tool such as DejaVu [7] could be used alongside our datarace detector to allow a full reconstruction of the conditions leading to the datarace, although DejaVu recording will incur an extra runtime overhead of about 30%.

Chapter 3

Datarace Detection Algorithm

Here we give the details of our datarace detection algorithm. Given a stream of access events from a running program (generated through instrumentation) as input, this algorithm tracks the accesses and raises an immediate flag when the most recent access is a datarace with a past access. The efficiency of the algorithm stems from our use of the fact that we do not guarantee the reporting of all dataraces in an execution. We use tries to efficiently represent and search past accesses and the *weaker-than relation* to ignore redundant accesses.

3.1 Weaker-Than Relation

Consider the case of a thread t writing memory location m twice, with no synchronization operations between the writes. If the second write to m races with an access to m by another thread t', then t's first write to m must also be a datarace with the access by t', since the locks held by t were identical for both writes. Since we only guarantee reporting one event involved in a datarace on m, our algorithm can safely ignore t's second write to m. In general, given past access events e_i and e_j, we say e_i is *weaker-than* e_j if and only if for all future accesses e_k, $IsRace(e_j, e_k)$ implies $IsRace(e_i, e_k)$. We call this relation *weaker-than* since intuitively, e_i is more weakly protected from dataraces than e_j (or equally protected). We exploit the *weaker-than* relation throughout our datarace detection system to greatly increase efficiency.

We can often determine that an event e_i is weaker than another event e_j solely by examining the information contained in each event (memory location, lockset, thread, and access type). First, we allow the the thread value of a past access event e to be the pseudothread $t_\perp$, meaning "at least two distinct threads". For some past event e_i, we set $e_i.t$ to $t_\perp$ when we see a later access event e_j such that $e_i.m = e_j.m$, $e_i.L = e_j.L$, and $e_i.t \neq e_j.t$. Setting $e_i.t$ to $t_\perp$ represents the fact that any future access e_k to $e_i.m$ such that $e_i.L \cap e_k.L = \emptyset$ must be a datarace (unless all accesses are reads), since multiple threads already accessed $e_i.m$ with lockset $e_i.L$. The use of $t_\perp$ decreases space consumption and simplifies our implementation, but it sometimes prevents us from reporting both threads that participate in a datarace.

We define a separate partial order $\sqsubseteq$ on threads (t_i and t_j), access types (a_i and a_j), and access events (e_i and e_j):

$$\begin{aligned}
t_i \sqsubseteq t_j &\iff t_i = t_j \vee t_i = t_\perp \\
a_i \sqsubseteq a_j &\iff a_i = a_j \vee a_i = \text{WRITE} \\
e_i \sqsubseteq e_j &\iff e_i.m = e_j.m \wedge e_i.L \subseteq e_j.L \\
&\qquad \wedge e_i.t \sqsubseteq e_j.t \wedge e_i.a \sqsubseteq e_j.a
\end{aligned}$$

Our algorithm detects when $e_i \sqsubseteq e_j$, which implies that e_i is *weaker-than* e_j, as is shown in the following theorem.

Theorem 1 (weaker-than). *For past accesses e_i and e_j and for all future accesses e_k, $e_i \sqsubseteq e_j \Rightarrow (IsRace(e_j, e_k) \Rightarrow IsRace(e_i, e_k))$.*

Proof. Assuming $(e_i \sqsubseteq e_j \wedge IsRace(e_j, e_k))$, we show that $IsRace(e_i, e_k)$ must be true. $e_i.m = e_j.m$ and $e_j.m = e_k.m$, which implies that $e_i.m = e_k.m$. $e_i.L \subseteq e_j.L$ and $e_j.L \cap e_k.L = \emptyset$, so $e_i.L \cap e_k.L = \emptyset$. Since $e_i.t \sqsubseteq e_j.t$ and $e_j.t \neq e_k.t$, $e_i.t \neq e_k.t$ (Note that since e_k is new, $e_k.t$ cannot be $t_\perp$). Finally, we have $e_i.a \sqsubseteq e_j.a$ and $e_j.a = \text{WRITE} \vee e_k.a = \text{WRITE}$. If $e_k.a = \text{WRITE}$, then e_i and e_k are clearly a datarace. If $e_k.a = \text{READ}$, then $e_j.a = \text{WRITE}$, and therefore $e_i.a = \text{WRITE}$, since $e_i.a \sqsubseteq e_j.a$. So, it is shown that $IsRace(e_i, e_k)$ is true, given our initial assumptions. □

If e_i is weaker than e_j, our datarace detector only stores information about e_i at most, saving in both time and space overhead[1]. We also use the weaker-than relation to filter many events so they never even reach the detector (discussed in greater detail in Chapters 4 and 5).

3.2 Trie-Based Algorithm

In this section, we delineate the actual execution of our trie-based datarace detection algorithm, which efficiently maintains an event history and compares new events with past events to check for dataraces.

3.2.1 Trie Data Structure

We represent relevant information about past access events using edge-labeled tries. Each memory location observed in some access event has an associated trie. The trie edges are labelled with lock identifiers, and the nodes hold thread and access type information for a set of access events (possibly empty). So, the lockset held for a particular memory access is represented by the path from the root to the node corresponding to the access in the trie corresponding to the access's location. Tries are space-efficient since locksets which share locks also share representation in the trie. To ensure this sharing property holds, our algorithm maintains the invariant that the lock labelling the edge whose destination is node n is always less than any lock labelling an edge whose source is n, under some total ordering on lock identifiers.

Some notation will be useful when describing our algorithm. Each node n in a trie has a thread field $n.t$, an access type field $n.a$, and a lock field $n.l$ corresponding to the lock identifier labelling the edge whose destination is n. Any inner nodes n' which do not represent any existing accesses are initialized with $n'.t \leftarrow t_\top$, where $t_\top$

[1]Note that in the infrequent case that our tool gives a spurious datarace report (because of our handling of array indices, for example), an optimization based on the weaker-than relation could lead to the suppression of a true datarace report while leaving the false positive. We believe this deficiency is minor (we did not encounter it with our test programs), and it can be overcome by using extra locking to suppress the false positive report and rerunning the detector.

means "no threads," and and $n'.a \leftarrow$ READ. Also, we define the meet operator $\sqcap$ for threads and access types:

$$\begin{array}{rll} \forall i. & t_i \sqcap t_i = t_i,\;\; t_i \sqcap t_\top = t_i,\;\; a_i \sqcap a_i = a_i & \\ \forall i.\forall j. & t_i \sqcap t_j = t_\bot & \text{if } t_i \neq t_j \\ \forall i.\forall j. & a_i \sqcap a_j = \text{WRITE} & \text{if } a_i \neq a_j \end{array}$$

3.2.2 Datarace Check

Given an access event e which has just executed, we first check if a past event e_p exists such that $e_p \sqsubseteq e$. If e_p does exist, then e can be ignored without affecting our reporting guarantees. We search for e_p by performing a traversal of the trie for memory location $e.m$, only traversing edges in $e.L$ (depth-first, in sorted order). This traversal guarantees that any nodes we encounter will meet the memory location and lockset requirements of the $\sqsubseteq$ relation. For each encountered node n, we check the condition $n.t \sqsubseteq e.t \wedge n.a \sqsubseteq e.a$. If this condition is true, then n represents the desired e_p. In almost all cases, this initial check allows us to ignore e, giving a large time savings.

If the aformentioned weakness check fails, we check if e is a datarace with any past event. Again, we traverse the trie corresponding to $e.m$, but this time we follow all edges. One of three cases holds for each node n encountered during this traversal:

Case I. $n.l \in e.L$. In this case, e shares the lock $n.l$ with all the accesses represented by n and its children. Therefore, e cannot be a datarace with any access represented by the subtrie rooted at n, and this branch of the trie need not be further traversed.

Case II. Case I does not hold, $e.t \sqcap n.t = t_\bot$, and $e.a \sqcap n.a =$ WRITE. In this case, we know n represents some past access e_p such that $e_p.m = e.m$, $e_p.t \neq e.t$, $e_p.a =$ WRITE $\vee$ $e.a =$ WRITE, and $e_p.L \cap e.L = \emptyset$, precisely the conditions of $IsRace(e_p, e)$. We immediately report e as a racing access and terminate the traversal.

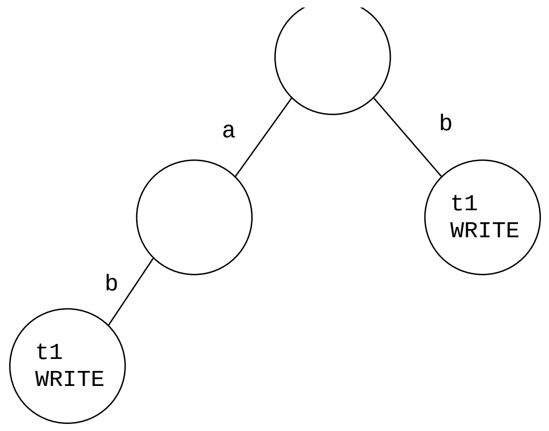


Figure 3-1: An example trie used in our datarace detection algorithm.

Case III. Neither Case I nor Case II holds. In this case, we traverse the children of n.

To make this checking step more concrete, consider the trie in Figure 3-1 representing accesses to some memory location m. The empty nodes do not correspond to any accesses, and the other nodes contain thread and access type information. Now, say that some thread $t2$ writes m with lockset $\{a\}$. To check this access, we first traverse the leftmost edge from the root node of the trie, labelled with lock a. After traversing this edge, Case I holds, so we do not need to traverse any further in this trie branch. We next traverse the edge labelled b from the root node, and we see that thread $t1$ has written m with lockset $\{b\}$. Now, Case II holds, and we immediately report that the most recent access by $t2$ is racing.

3.2.3 History Update

If e does not race with any past events, we next update the trie for $e.m$ to reflect e's information by performing another traversal following the edges with locks in $e.L$. If a node n already exists which represents accesses with lockset $e.L$, we update n so that $n.t \leftarrow n.t \sqcap e.t$ and $n.a \leftarrow n.a \sqcap e.a$. If no such n exists, we add the necessary edges and inner nodes to create n, setting $n.t \leftarrow e.t$ and $n.a \leftarrow e.a$.

It is possible that some past accesses stored in the trie for $e.m$ are weaker than

e, and after we add e's information to the trie, we perform a final pass to remove information about such accesses. We traverse the entire trie, searching for nodes n' such that the accesses represented by n' were performed with a lockset that is a superset of $e.L$. For each such node n', we also check the condition $e.t \sqsubseteq n'.t \wedge e.a \sqsubseteq n'.a$. If the condition holds, then n' represents accesses which are stronger than e, so we re-initialize the node, setting $n'.t \leftarrow t_\top$ and $n'.a \leftarrow$ READ. We then traverse the children of n' in a depth-first manner. Finally, if after this pass, n' and all of its children have been re-initialized (and therefore represent no accesses), we can prune n' and its children from the trie.

3.2.4 Analysis

Here we give an analysis of the worst-case time and space complexity of our datarace detection algorithm. Let N_{max} be the maximum number of nodes in a trie for a given execution of a program. Then, the space required to store the tries is $O(N_{max}M)$, where M is the number of unique memory locations accessed by the program. Since each access event requires a constant number of trie traversals, the worst-case time complexity of our algorithm is $O(N_{max}E)$, where E is the number of access events in the execution. Finally, we can bound N_{max} by the maximum number of leaf nodes in any trie multiplied by the deepest path in any trie. The deepest path is bounded by the size of the largest lockset held during the execution, which we will call L_{max}. The maximum number of distinct locksets held while accessing a given memory location, S_{max}, bounds the maximum number of leaf nodes in a trie. So, N_{max} can at most be $S_{max}L_{max}$. It should be noted that while we believe our datarace detection algorithm is efficient, our results indicate that the key factors contributing to our low overhead are the static and dynamic optimizations which allow us to completely avoid executing our datarace detection code (discussed in Chapters 4 and 5). In some sense, if we are forced to run the steps described in this chapter for a given event, then our performance edge for that event has already been lost.

3.3 Implementation

Our datarace detector is implemented entirely in Java for the Jalapeño Research Virtual Machine [2] (now known as the Jikes Research Virtual Machine). Jalapeño is a high-performance virtual machine designed for servers, written almost entirely in Java with no separate virtual machine necessary for execution. Important features of Jalapeño include a user-level thread scheduler which multiplexes Java threads onto operating system threads, a modular garbage collection architecture with several different collection algorithms supported, and an adaptive optimization framework which selectively applies different optimizations to methods depending on their execution frequency. The clean and modular design of Jalapeño greatly facilitated the implementation of our datarace detector. See [2] for a more detailed description of Jalapeño's runtime and compilation systems.

For the most part, our implementation is straightforward, with the datarace detection algorithm running alongside the target program. The instrumentation which generates the input stream of access events for our algorithm is discussed in detail in Chapter 5. We use memory addresses to identify logical memory locations, which can introduce some nasty subtleties in interactions with garbage collection. A copying garbage collector could move objects to different addresses, which would force us to track the object copying and update our identifiers appropriately. Even a non-copying collector can cause serious problems by allocating an object at some address, garbage collecting that object, and then allocating a new object at the address. This behavior could also be handled by our algorithm through proper flushing of state associated with garbage-collected objects and the use of identifiers distinct from memory addresses for locks. However, for our prototype implementation, we avoid these problems by using a large enough heap during execution that garbage collection never occurs.

One extra space optimization we perform is *trie packing*, which allows us to use one trie to represent up to 32 distinct memory locations. Each trie node's state is implemented with three 32-bit words, and each memory location has an associated index to extract its information from the fields of a node. Two of the words represent the access type for each location (whether it has been accessed and whether it has been written), and one word represents whether each location has been accessed by many threads (ie. whether $n.t = t_\perp$ for each location). We always assign new memory locations to the most recently created trie, the idea being that locations which are accessed together are often accessed by the same thread and with the same locks. Each node can only hold one thread id for all of its memory locations (although each memory location can have its $t_\perp$ flag set independently), so in some cases we cannot represent all the necessary access information for two different memory locations in one trie. In this case, we evict one of the locations from the trie and assign it to the most recent trie as if it were a new memory location. Occasionally, another conflict occurs, in which case we create a new trie for the memory location.

Chapter 4

Dynamic Optimization

The datarace detection algorithm described in Chapter 3 takes as input a stream of access events from a running program. The largest performance improvements in our datarace detection system stem from optimizations which greatly decrease the number of access events which need to be reported to the detector. In this chapter, we describe the dynamic optimization used to filter access events, a caching scheme for quickly detecting if an access can be ignored.

4.1 Cache Design

In Chapter 3, we defined the *weaker-than* relation and showed how it could be used as the basis for space and time optimizations in our datarace detection algorithm. We observed that in practice, almost all dynamic accesses performed by a program are ignored because some previous stored access is weaker. Therefore, the critical path in our datarace detector for a new access was the initial check for a past weaker access, not the check to see if the access was involved in a datarace. Caching seemed like a natural way to significantly improve the performance of the weakness check.

At a high level, our caching scheme works as follows. We cache information about recent accesses that have been sent to the datarace detector for checking, indexed by memory address. When a new access e of memory address m occurs, we first look up m in our cache. We maintain the invariant that if an entry for m exists in the cache, an access of m has already occurred which is weaker than e, so e can be ignored. If there is no entry for m in the cache, then e is sent to the datarace detector for a full check.

Our cache policy must ensure that if an entry for memory address m is found in the cache, then any future access m must have a corresponding past weaker access. Recall the conditions used to detect that access e_i is weaker than access e_j:

- $e_i.m = e_j.m$,
- $e_i.L \subseteq e_j.L$,
- $e_i.t \sqsubseteq e_j.t$, and
- $e_i.a \sqsubseteq e_j.a$.

Since our caches are indexed by memory address, the first condition clearly holds. To satisfy the third and fourth conditions, we keep separate read and write caches for each thread. So, for some event e, if a lookup of $e.m$ results in a cache hit, we know a past event e_p occurred such that $e_p.m = e.m$, $e_p.t = e.t$, and $e_p.a = e.a$.

Our techniques for ensuring the second condition, that events represented in the cache occurred with a subset of the current lockset, are a bit more complex. We monitor the set of locks held by each thread, and when a thread t releases a lock l, we evict all events e from the t's read and write caches such that $l \in e.L$. Note that the execution of a `monitorexit` by a thread does not necessarily correspond to the release of a lock, since in Java a lock can be acquired more than once by a thread. We track the number of times a lock has been acquired by a thread, and only evict entries from the cache when the lock is truly relased.

To efficiently evict cache entries corresponding to events executed by a thread while holding a particular lock, we exploit the strict nesting of Java's monitors. The Java source language guarantees that the most recent lock acquired by a thread will be the first to be released by the thread[1]. So, for each lock l held by thread, we need only track the entries in the cache for which l was the most recently acquired

[1]Note that this guarantee does not necessarily hold for Java bytecodes. We assume that target programs have been compiled from Java source.

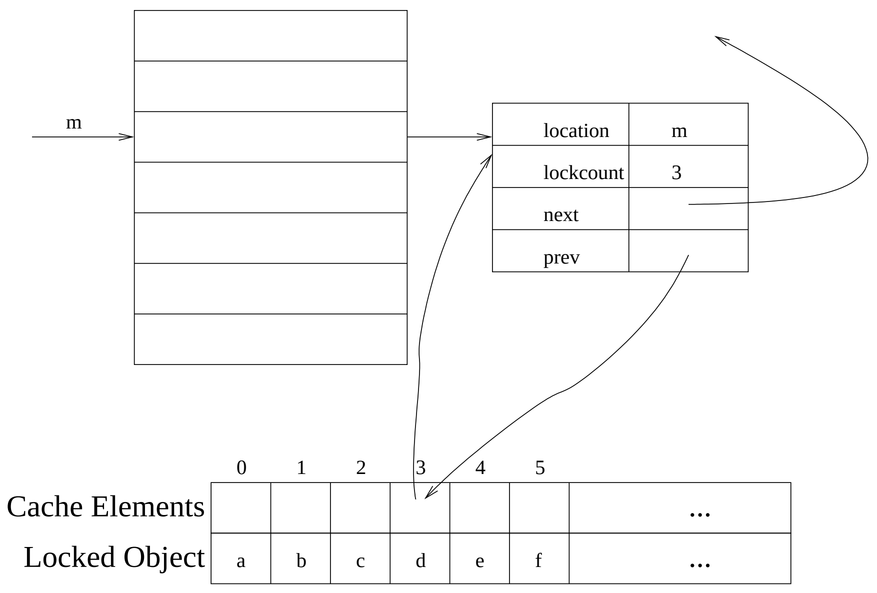


Figure 4-1: The cache.

lock when the corresponding event occurred. We maintain this information with a doubly-linked list for each lock held by the thread. The doubly-linked list allows for efficient eviction of all entries on the list or of a single entry in the case of a cache conflict.

Figure 4-1 gives a view of the organization of our cache. The beginning of the doubly-linked list for lock d is shown, and the first entry of the list is expanded. The cache entry contains the memory location m, the next and previous pointers of the doubly-linked list for entries added while d was the most recently acquired lock, and a lockcount entry corresponding to the depth of the lock nesting of d. The lockcount entry could be used for further optimizing the case where a single lock l is being acquired and released repeatedly. In this case, the cache entries for l need not be flushed every time it is released, but only when a different lock l' is acquired at its locking depth. In our implementation, the cache hit rate was high enough that it did not seem like this extra optimization would be useful, so it was not implemented, but the lockcount field remains to allow for a future implementation.

4.2 Implementation

For each thread, we maintain two 256-entry direct mapped caches (a read cache and a write cache), indexed by memory address. Our cache design and the Jalapeño infrastructure allowed us to implement the cache lookup path very efficiently. Since we maintain the property that all entries in the caches represent events that are weaker than any new event (assuming memory addresses match), the cache lookup solely involves finding an entry which matches the current event's memory address. Thread, access type, and lockset information are not stored explicitly in the cache. Our hash function simply multiplies the memory address by a large constant and then uses the high-order 16 bits of the result, which seems to give few conflicts in practice and can be computed efficiently (without division).

Our cache is implemented in Java with some Jalapeño-specific techniques to increase performance. We use `VM_Magic` calls [2] to tell Jalapeño not to perform null pointer and array bounds checks in our cache lookup code. We also force Jalapeño's optimizing compiler to always inline the cache lookup code into the user program. Through these efficiency tweaks, we decreased the cost of the cache hit path to ten PowerPC instructions.

Chapter 5

Static Optimization

Here we describe our techniques for statically decreasing the cost of datarace detection. The first optimization identifies statements which will never be involved in a datarace when executed. The second optimization detects statements which if instrumented would always produce access events for which another weaker event exists. The statements flagged by these two optimizations need not be instrumented for datarace detection, leading to a smaller dynamic set of access events which need to be checked by our datarace detection.

5.1 Static Datarace Analysis

Our static datarace analysis computes a *potential datarace set*, a conservative set of statement pairs whose executions could be a datarace. Statements which do not appear in any pair in this set cannot be involved in a datarace in any execution of the program, and therefore do not need to be instrumented for dynamic datarace detection. Our analysis goes beyond escape analysis of accessed objects [8, 27], performing an interthread control flow analysis and points-to analysis of thread, synchronization, and access objects. We will only briefly describe our static datarace analysis techniques here, as I did not play a role in the development of the techniques. A more detailed presentation of these analyses can be found in [10].

5.1.1 Example

We first present an example, seen in Figure 5-1, to illustrate some important properties of a static datarace analysis. In this example, `MainThread` creates and starts two instances of `ChildThread`, and all threads write various fields of `Obj` objects. Given such a program as input, our static datarace analysis identifies a set of pairs of statements which could be involved in dataraces, attempting to make this set as small as possible while still being conservative.

Points-to analysis plays a key role in identifying whether statement pairs can be involved in dataraces. For example, if the `x` field of `MainThread` can point to the same object as the `b` field of the `ChildThread` created in statement `S11`, then the statement pair (`S18`, `S20`) must be added to the potential datarace set (the child thread invokes `MainThread.m1` in statement `S32`). A traditional *may* points-to analysis identifies this possibility. Now, consider the two invocations of `MainThread.m1` in statements `S16` and `S32`. If the `p` field of `MainThread` and the `a` field of some `ChildThread` can point to *different* objects, then these two invocations may not be properly synchronized, and (`S20`, `S20`) must be added to the potential datarace set. To identify this case, a *must* points-to analysis must be employed, and the result negated.

Escape analysis can help to quickly eliminate some statements from consideration in the static datarace analysis. For example, consider statement `S42`. An escape analysis shows that the object pointed to by `z` cannot escape method `m2`, and therefore must be a *thread-local* object. Therefore, statement `S42` cannot be involved in dataraces. The escape analysis utilized is fully described in [8].

5.1.2 Static Datarace Conditions

In Chapter 2, we defined a datarace between two accesses in terms of their thread, access type, memory location, and lockset information. Given two statements x and y, our datarace definition can be formulated conservatively for static analysis as follows:

$$\begin{aligned} IsMayRace(x,y) \Longleftarrow\ & \mathit{AccessesMayConflict}(x,y) \wedge \\ & (\neg \mathit{MustSameThread}(x,y)) \wedge (\neg \mathit{MustCommonSync}(x,y)) \end{aligned} \tag{5.1}$$

```
// thread main
class MainThread {
   static Obj p, q, x;
   public static void main(String args[]) {
      . . .
S11:  Thread T1 = new ChildThread(...);
S12:  Thread T2 = new ChildThread(...);
S13:  T2.start();
S14:  T1.start();
      . . .
S15:  synchronized(p) {
S16:     m1(q);
      } // synchronized
      . . .
S17:  T2.join(); // wait until T2 terminates
S18:  x.f = 200;
   }

   public static void m1(Obj y) {
S20:  y.f = 100;
   }
} // class MainThread

// thread T1, T2
class ChildThread implements Runnable {
   Obj a, b, c;
   public void run() {
S30:
S31:  synchronized (a) {
S32:      MainThread.m1(b);
S33:      m2(c);
      } // synchronized
S34:
   }

   public void m2(Obj w) {
S40:
S41:  Obj z = new Obj(...);
S42:  z.f = ...
   }
} // class ChildThread
```

Figure 5-1: Example program for static datarace analysis.

$AccessesMayConflict(x, y)$ = *true* if x and y could possibly access the same memory location and either x or y writes the location. We use information from a *may* points-to analysis to determine this condition. This analysis would determine whether `MainThread.x` and `ChildThread.b` (for some `ChildThread` instance) can be aliased in the program from Figure 5-1, for example, in checking whether statements `S18` and `S20` can race.

$MustSameThread(x, y)$ = *true* if and only if x and y are *always* executed by the same thread. To compute this condition, we use *must* points-to information for thread objects. For Figure 5-1, this analysis would determine that $MustSameThread(\texttt{S18}, \texttt{S20}) = false$ since `S20` can be executed by child threads `T1` and `T2`.

$MustCommonSync(x, y)$ = *true* if and only if x and y are *always* executed while holding some common synchronization object. We use *must* points-to information for synchronization objects to compute this condition. The computation of this condition would determine whether the synchronization objects used in statements `S15` and `S31` from Figure 5-1 are always the same, for example.

5.1.3 Interthread Control Flow Graph

The *interthread control flow graph* (ICFG) is a detailed interprocedural representation of a multithreaded program used in performing static datarace analysis. Nodes in the ICFG represent instructions, with distinguished *entry* and *exit* nodes for methods and synchronized blocks. Edges represent four types of control flow: *intraprocedural*, *call*, *return*, and *start*. The first three types of edges are found in a standard interprocedural control flow graph and are referred to as *intrathread edges*. A *start* edge is an *interthread edge* from an invocation of the method `Thread.start()` to the corresponding `run()` method which executes in the new thread. The target entry node of a start edge is called a *thread-root node*. An ICFG path which only includes intrathread edges is an *intrathread path*, while a path which includes some interthread edge is an *interthread path*. The ICFG for the program from Figure 5-1 can be seen in Figure 5-2(A) (dashed edges are start edges).

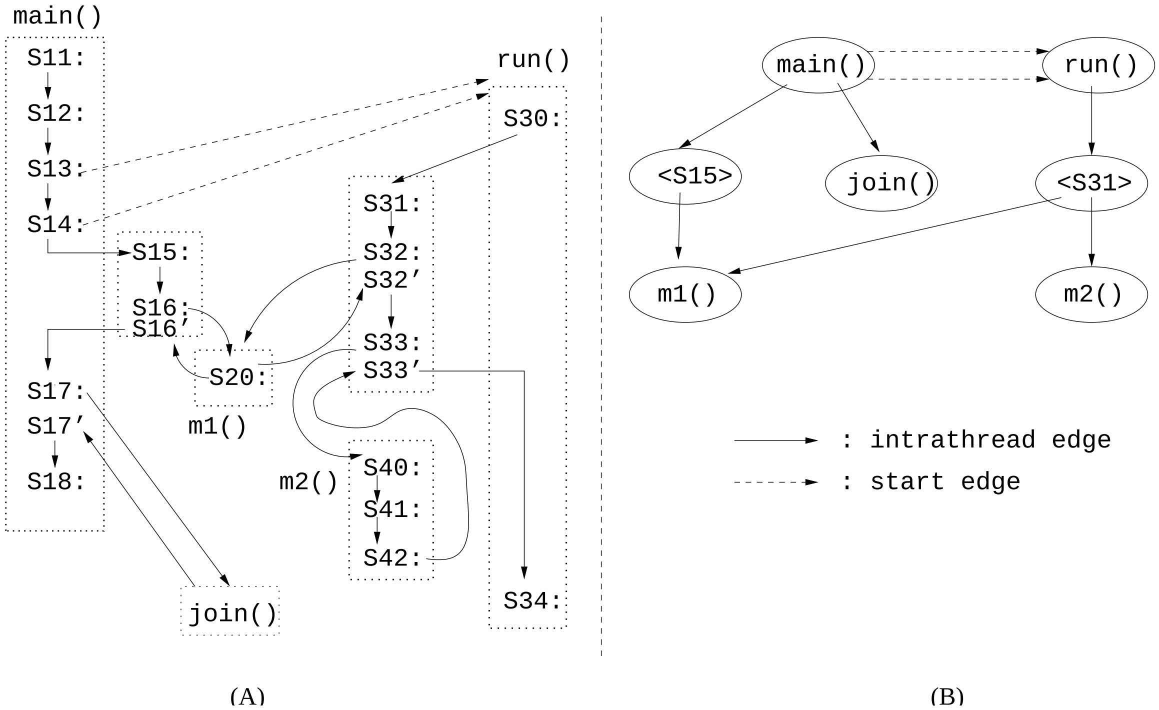


Figure 5-2: Interthread Control Flow Graph (A) and Interthread Call Graph (B) of the Example Program in Figure 5-1.

For scalability when analyzing large programs, an abstraction of the ICFG called the *interthread call graph* (ICG) is utilized. An interthread call graph is similar to a standard call graph. Distinguishing features of the ICG are the start edges seen in the ICFG and separate nodes for synchronized blocks. Figure 5-2(B) shows the ICG for the program from Figure 5-1. The nodes from the ICFG which have been combined are indicated by dashed boxes in Figure 5-2(A).

5.1.4 Points-To Analysis

Here we describe how we use points-to analysis to compute the conditions described in section 5.1.2. Our points-to analysis is a flow-insensitive, whole program analysis which operates as follows. For each allocation site of the program, a unique *abstract object* is created, representing all the concrete objects created at the allocation site at runtime. We compute the set of possibly referenced abstract objects for each access in the program. So, if for two access statements, the intersection of the abstract object set associated with each statement is non-empty, the statements *may* access the same memory location at runtime.

A more expensive analysis is necessary to compute precise must points-to information for a program. We employ a simple but very conservative must points-to analysis based on examining statements which can execute at most once in a given execution, *single-instance* statements. If an allocation site is a single-instance statement, its associated abstract object is a *single-instance* object. An access statement which can only reference one single-instance abstract object is associated with the object by the must points-to relation.

Let the must and may points-to sets for statement x be $MustPT(x)$ and $MayPT(x)$, respectively. Then, $AccessesMayConflict(x, y)$ from Equation 5.1 can be computed as follows:

$$\begin{aligned} \mathit{AccessesMayConflict}(x,y) &= (MayPT(x) \cap MayPT(y) \neq \emptyset) \\ &\wedge\ (field(x) = field(y)) \\ &\wedge\ (IsWrite(x) \vee IsWrite(y)), \end{aligned} \tag{5.2}$$

where $field(x)$ is the field accessed by x and $IsWrite(x) = true$ iff x is a write.

To compute $MustSameThread(x, y)$, we use must points-to information and the program's ICFG. Let $ThStart(u)$ be the set of thread-root nodes in the ICFG such that for all $v \in ThStart(u)$, access u is reachable from v through an intrathread ICFG path. Then, the following equations will compute $MustSameThread(x, y)$:

$$\begin{aligned} MustThread(u) &= \bigcap_{v \in ThStart(u)} MustPT(v.\texttt{this}) \\ MustSameThread(x,y) &= \\ &(MustThread(x) \cap MustThread(y) \neq \emptyset), \end{aligned} \tag{5.3}$$

where $v.\texttt{this}$ denotes the `this` pointer of thread-root node v.

Finally, $MustCommonSync(x, y)$ can be computed using the program's ICG and must points-to information. We first compute $MustSync(v)$, the set of abstract objects which must be locked for any execution of v. For each node n in the ICG, let $Synch(n) = true$ if and only if n represents a synchronized method or block. Also, let u_n be the statement which accesses the synchronization object for n if

$Synch(n) = true$, and let $Pred(n)$ be the set of *intrathread* predecessors of n in the ICG. $MustSync(v)$ can be found by computing the following set of dataflow equations:

$$\begin{aligned}
Gen(n) &= \begin{cases} MustPT(u_n) & \text{if } Synch(n) \\ \emptyset & \text{otherwise} \end{cases} \\
SO_o^n &= SO_i^n \cup Gen(n),\ SO_i^n = \bigcap_{p \in Pred(n)} SO_o^p \\
MustSync(v) &= SO_o^n, \forall v \in n.
\end{aligned}$$

Given the $MustSync$ relation, we can now easily compute $MustCommonSync(x, y)$:

$$\begin{aligned}
MustCommonSync(x, y) &= \\
&(MustSync(x) \cap MustSync(y) \neq \emptyset).
\end{aligned} \tag{5.4}$$

Combining results from Equations 5.2, 5.3, and 5.4, we compute the $IsMayRace$ condition from Equation 5.1.

5.1.5 Extending Escape Analysis

Escape analysis typically aims to identify those objects which are accessed by exactly one thread during any execution of a program, often called *thread-local* objects. Since thread-local objects cannot be involved in dataraces, accesses to thread-local objects do not need to be instrumented for datarace analysis, often giving a large savings in runtime overhead.

In some cases, an object is not strictly thread-local as defined above, but in a sense the object still does not “escape.” For example, when a child thread performs some computation, the state of the computation is often held in fields of the child thread (which extends `java.lang.Thread`) which were initialized by the parent thread at construction time. The objects referenced by these fields are not thread-local, since the parent thread initialized them, but in many cases the objects still cannot be involved in dataraces. To identify this case statically and therefore further decrease the cost of dynamic datarace detection, we introduces the notion of a *thread-specific* object.

Thread-specific objects can be formally defined as follows. If an object o is thread-local to thread t, then it is also thread-specific to t. Also, o is thread-specific to t if every access to o is done by t *or* by the parent thread p that constructs t, such that p accesses o only before the completion of t's construction and t's construction and running do not overlap. We have implemented an approximate algorithm for finding thread-specific objects, and we use this information to exclude statements which only access thread-specific objects from any statement pair in the potential datarace set. We plan to do more work to generalize the notion of thread-specificity and develop better algorithms for statically detecting such conditions.

5.2 Instrumentation Elimination

Our second compile-time optimization technique stems from the weaker-than relation defined in Chapter 3. If the access events generated by instrumentation for a given statement will always be ignored because of other weaker events, there is no need to instrument that statement. In this section, we describe how we extend the weaker-than relation to instrumentation statements and employ a loop peeling transformation to avoid this sort of unnecessary instrumentation.

5.2.1 Static Weaker-Than Relation

The static weaker-than relation extends the notion of weaker-than to instrumentation for program statements, and can be defined as follows. Given an instrumentation statement S, let $Events(S)$ be the set of access events generated by S in a given execution. Also, we define $Exec(S_i, S_j)$ as follows:

Definition 1. For statements S_i and S_j, $Exec(S_i, S_j)$ is true iff (1) S_i is on every intraprocedural path starting at method entry containing S_j and (2) no method invocations exist on any intraprocedural path between S_i and S_j.

The first part of the definition represents the condition that for each execution of statement S_j, there exists a corresponding execution of statement S_i, and the second part conservatively avoids the case where a new thread could be started with a `start()` invocation between S_i and S_j. The static weaker-than relation on instrumentation statements is defined as follows:

Definition 2. S_i is *weaker-than* S_j, written as $S_i \sqsubseteq S_j$, iff in all possible executions, $Exec(S_i, S_j) \wedge \exists e_i \in Events(S_i) \,|\, \forall e_j \in Events(S_j) \,|\, e_i \sqsubseteq e_j$, with the relation $e_i \sqsubseteq e_j$ defined in Chapter 3.

Computing the condition $S_i \sqsubseteq S_j$ for arbitrary S_i and S_j would require a complex and expensive interprocedural analysis. Instead, we employ a conservative intraprocedular analysis for determining $S_i \sqsubseteq S_j$ when S_i and S_j belong to the same method.

We model the instrumentation of an access instruction with the pseudo-instruction $trace(o, f, L, a)$. o is the accessed object, f is the field of o being accessed, L is the set of locks held during the access, and a is the access type (READ or WRITE). If the access is to a static field, then o represents the class in which the field is declared, and if the access is to an array, then f represents the array index. All operands of a *trace* instruction are *uses* of their values. Given no other information, we insert *trace* instructions after every memory access statement in the program. If the static datarace analysis from Section 5.1 has been run, we use its output to only instrument instructions which can possibly be involved in a datarace. Also note that no thread information is represented in *trace* instructions, since we do not optimize across thread boundaries and information about the current thread is always available at runtime.

After *trace* instructions have been appropriately inserted, we attempt to eliminate some of them using the static weaker-than relation. Given *trace* statements $S_i = trace(o_i, f_i, L_i, a_i)$ and $S_j = trace(o_j, f_j, L_j, a_j)$, the condition $S_i \sqsubseteq S_j$ can be computed as the conjunction of easily verifiable conditions (notation will be explained):

$$\begin{aligned} S_i \sqsubseteq S_j \quad \Longleftarrow \quad & dom(S_i, S_j) \wedge a_i \sqsubseteq a_j \wedge outer(S_i, S_j) \\ & \wedge valnum(o_i) = valnum(o_j) \wedge f_i = f_j . \end{aligned}$$

To approximate the $Exec(S_i, S_j)$ condition, we use the *dominance relation* between program statements [19]. $dom(S_i, S_j) = true$ if and only if S_i is on *every* program path from the entry of the method to S_j. We also need to show that S_i will always produce some access event e_i weaker than all events e_j produced by S_j. Recall the conditions used to check that $e_i \sqsubseteq e_j$: $e_i.t \sqsubseteq e_j.t$, $e_i.a \sqsubseteq e_j.a$, $e_i.L \subseteq e_j.L$, and $e_i.m = e_j.m$. Since our analysis is intraprocedural, we know $e_i.t = e_j.t$, and a_i and a_j can be used to directly check that $e_i.a \sqsubseteq e_j.a$. We use the nesting of Java's synchronization blocks to verify $e_i.L \subseteq e_j.L$, verifying that S_j is at the same nesting level of synchronization blocks as S_i *or* nested within S_i's block (written $\mathit{outer}(S_i, S_j)$). Finally, to check the condition $e_i.m = e_j.m$, we verify that $\mathit{valnum}(o_i) = \mathit{valnum}(o_j)$, where $\mathit{valnum}(o_i)$ is the *value number* of object reference o_i [3], *and* that $f_i = f_j$.

5.2.2 Implementation

Our instrumention insertion and elimination passes are implemented as part of the Jalapeño optimizing compiler infrastructure. We created a new high-level intermediate representation (HIR) instruction corresponding to *trace*, and a pass inserts them into each method as described above. After insertion, the HIR representation is converted to *static single assignment* (SSA) form, and the dominance relation is computed [3]. Then, the elimination of *trace* instructions based on the static weaker-than relation is perfomed, utilizing information an existing global value numbering phase. Note that in our implementation, *trace* pseudo-instructions are modelled as having an unknown side effect so that they are not incorrectly eliminated by the optimizing compiler as dead code.

In a later phase (still operating on HIR), *trace* pseudo-instructions are expanded to calls of datarace detection methods. We force Jalapeño to inline these calls for efficiency. After inlining, some general optimizations such as constant propagation are again applied to further improve the performance of the instrumentation. Finally, the HIR is converted to lower-level intermediate forms and eventually assembled by the compiler, with no further instrumentation-specific optimization.

5.2.3 Loop Peeling

Multiple executions of a loop body can often generate many ignorable access events. Consider the loop from statement `S10` to statement `S13` in Figure 5-3. The first execution of `S13` generates a non-redundant access event, but in later loop iterations, the events generated by `S13` will be stronger than the event generated in the first iteration, and can therefore be ignored. However, statically eliminating the generation of the redundant events is non-trivial. `S13` cannot be eliminated completely using our existing analysis based on static weaker-than, since in the first iteration of the loop its generated access event is non-redundant. We also cannot apply the standard loop-invariant code motion technique to move `S13` outside the loop. Statement `S11` is a *potentially excepting instruction* (PEI), meaning it can throw an exception which will cause an immediate exit from the loop body. PEI's appear frequently in Java because of null-pointer and array bounds checks. If `S11` throws an exception in the first execution of the loop body, then `S13` would never execute, so moving `S13` outside the loop is unsafe.

To handle loops which generate many redundant access events, we perform *loop peeling* on the program. Loop peeling transforms a loop so that a copy of the loop body executes in the first iteration, while the remaining iterations execute the original loop body. The effects of loop peeling combined with our existing pass to eliminate redundant instrumentation can be seen in statements `S20` through `S26` of Figure 5-3. Statement `S20` is a guard for the case in which the loop body never executes, and the loop condition in statement `S24` is modified appropriately to account for the execution of the first loop iteration in statements `S21` through `S23`. The same exception handler is utilized for statements `S21` and `S25`. After loop peeling, the *trace* statement inside the loop body can be eliminated since the *trace* statement in the peeled copy is statically weaker. Therefore, the write to $a.f$ is reported at most once after the transformations, achieving the goal of statically eliminating all redundant event generation from the loop.

Our loop peeling transformation is currently implemented as a bytecode to byte-

```
// Before optimization.
S00: A a;
S10: for(...) {
S11:   PEI
S12:   a.f = ...;
S13:   trace(a, f, L, W)
     }

// After optimization. Redundant trace
// statements have been eliminated.
S20: if (...) {
S21:   PEI
S22:   a.f = ...;
S23:   trace(a, f, L, W);
S24:   for (...) {
S25:     PEI
S26:     a.f = ...;
       }
     }
```

Figure 5-3: Example of Loop Peeling Optimization

code transformation which is applied before all other phases. The implementation naïvely peels all loops, which results in a code size increase exponential in the loop nesting depth. We plan to address this issue in the future by using profiling to only peel “hot” loops which have the greatest impact on performance.

5.3 Interactions with Ownership Model

The ownership model we use to handle the orderings implied by thread creation, discussed in section 2.1.4, can interact in subtle ways with optimizations based on the weaker-than relation. Here we discuss in more detail how the ownership model is implemented and what effects it has on the guarantees of our race detector while our other optimizations are enabled.

To track memory locations which are in the owned state, only accessed by one thread, we maintain for each location information about its owner thread. When a location m is first accessed by a thread t, we sets its owner thread to t, and we do not consider the initial access for dataraces. Any subsequent accesses to m by t while it remains the owner thread are also not considered for datarace detection. When some different thread t' accesses m, we set the owner thread information for m to $\perp$, indicating that m has been accessed by multiple threads, and we begin to track accesses to m for datarace detection (including the initial access of t'). At this point, m is in the shared state, and all subsequent accesses to m are checked for dataraces.

Unfortunately, our definitions of the weaker-than relation do not consider the behavior of the ownership model. Therefore, it is possible that even if event e_i is weaker than event e_j, it is not safe to eliminate e_j if we want to maintain our reporting guarantees. The bad case occurs when $e_i.m$ is in the owned state when e_i occurs but changes to the shared state for when e_j occurs, since in this case e_j could be a datarace with another access while e_i cannot be.

For the dynamic cache discussed in Chapter 4, this boundary case can be handled in a straightforward manner by flushing all cache entries for a location m when it changes from the owned state to the shared state. Unfortunately, fixing the instrumentation elimination pass based on the static weaker-than relation is not as simple. Statically, it is very difficult to prove that between two statements S_i and S_j such that $S_i \sqsubseteq S_j$, the accessed memory location m cannot change from owned to shared dynamically. The only way to completely eliminate instrumentation in a safe way is to use *post-dominators* instead of dominators in approximating our $Exec(S_i, S_j)$ condition. Using post-dominators, the weaker statement S_i would occur later in the method, and we would eliminate instrumentation for the earlier statement S_j. However, post-domination is an extremely weak notion in Java, since almost all bytecode instructions can throw an exception and therefore no guarantees can be made about the execution of subsequent statements.

Our implementation actually ignores the interactions between weaker-than based optimizations and the ownership model, meaning that the possibility exists of our datarace detector failing to report bugs because it has unsafely suppressed instrumentation or accesses. This potential flaw is ameliorated by two considerations. First, we ran all of our test programs with and without the unsafe optimizations several times, and no new dataraces were reported with the optimizations disabled, indicating that this problem may be very minor in practice. Second, Chapter 6 will show that while the instrumentation elimination optimization gives an important improvement for one scientific benchmark, in general it does not seem to be as effective as the static datarace analysis or the dynamic cache in reducing overhead for the benchmarks written in a more object-oriented style. So, if the type of false negatives described in this section were actually a serious concern, it seems likely that the instrumentation elimination could just be disabled with a relatively mild effect on performance.

Chapter 6

Results

Here we present results for a preliminary implementation of our datarace detection algorithm and optimizations, applied to several test programs. Our results show that our methods are both efficient, with overhead ranging from 13% to 42% for our tests, and precise, with almost all reported dataraces being truly unsynchronized accesses.

6.1 Test Programs

Table 6.1 gives information about our test programs. `mtrt` is part of the standard SPECJVM98 benchmark suite, and the other tests were obtained from the authors of [23]. `sor2` is a modified version of the original `sor` benchmark, with some loop-invariant expressions for calculating array subscripts manually hoisted out of inner loops. The modified program is semantically equivalent to the original, and the optimizations could have been performed automatically by a compiler with an intraprocedural analysis. However, Jalapeño did not implement the optimization, and our performance is significantly affected by its application. Also, `elevator` has been slightly modified to terminate when its computation completes rather than its original behavior of just hanging. Finally, note that `elevator` and `hedc` are interactive benchmarks and therefore not CPU-bound, so we do not report any performance overheads for them.

Example	Lines of Code	Num. Dynamic Threads	Description
`mtrt`	3751	3	MultiThreaded Ray Tracer from SPECJVM98
`tsp`	706	3	Traveling Salesman Problem solver from ETH [23]
`sor2`	17742	3	Modified Successive Over-Relaxation benchmark from ETH [23]
`elevator`	523	5	A real-time discrete event simulator
`hedc`	29948	8	A Web-crawler application kernel developed at ETH [23], using a concurrent programming library by Doug Lea.

Table 6.1: Benchmark programs and their characteristics

6.2 Accuracy

Table 6.2 indicates the number of races reported by our datarace detection algorithm and some of its variants. Note that while we normally report races for each relevant field of an object, the table lists only the number of distinct objects for which races are reported, for comparison purposes. The "Detected" column gives the number of dataraces reported by our full algorithm, and the "True" column indicates how many of these objects were actually accessed in an unsynchronized manner. The "FieldsMerged" column lists the number of objects on which we would report races if we did not distinguish their fields (as in [23]). Finally, if we disabled our ownership model for approximating the orderings enforced by `Thread.start()`, we would report races on the number of objects indicated in the "NoOwnership" column.

Nearly all the races we report with our full detection algorithm correspond to truly unsynchronized accesses to shared memory. In tests for which this is not the case (`tsp` and `sor2`), higher-level synchronization which our lockset-based datarace model does not capture is utilized. `tsp` uses synchronized shared queues (producer-consumer), and `sor2` uses barrier synchronization. As is suggested in Section 2.1.3, a datarace detector based on the happened-before relation would recognize these higher-

Example	Detected	True	FieldsMerged	NoOwnership
`mtrt`	2	2	2	12
`tsp`	5	1	20	241
`sor2`	4	0	4	1009
`elevator`	0	0	0	16
`hedc`	5	5	10	29

Table 6.2: Number of Objects With Races Reported

level synchronizations and would not report the false dataraces seen in our method. However, our data indicates that this problem is small, and our low performance overhead seems to justify our slightly decreased precision.

6.2.1 Detected Dataraces

Two dataraces are reported for `mtrt`. The first is on a field whose value is not used, `RayTrace.threadCount`, and therefore does not affect correctness. The second race is on the `ValidityCheckOutputStream.startOfLine` field in the SPEC test harness, and it could lead to incorrect line breaks in the output.

The `tsp` test program has a serious datarace on its `TspSolver.MinTourLen` field which could lead to the reporting of incorrect values in its output. This datarace was reported in previous work [23].

The dataraces reported for `hedc` have two main causes. A field containing the size of a thread pool is accessed without appropriate locking, but this datarace does not affect the correctness of the program. The second cause, unsychronized accesses to the `Task.thread_` field, is more serious, and could lead to a `NullPointerException` with the appropriate thread schedule. This datarace is very subtle and would be extremely difficult to find with normal testing and debugging. Previous work [23] mistakenly characterized this datarace as benign, indicating the trickiness of the bug and the need for precision in a datarace detection tool (their tool reported over 100 dataraces for the same test case).

6.2.2 Datarace Definition Comparison

The “FieldsMerged” column of Table 6.2 indicates the practical difference between the common lock datarace definition we use and object races, previously discussed in Section 2.1.2. The numbers in the “FieldsMerged” column are a lower bound on the number of object races in the test runs, since we do not consider the effects of treating method calls as writes. For `hedc`, not distinguishing fields leads to spurious datarace reports for the `LinkedQueue` class, which has some fields accessed with synchronization and others which are not. We also report spurious races for the `MetaSearchRequest` class, in which some fields are thread-local and others are accessed by multiple threads. Given the mental effort required to investigate even one datarace report and understand the associated program behavior, we feel that the extra precision of the common lock datarace definition is practically necessary for making a useful datarace detection tool.

The only practical difference we observed between the common lock datarace definition and the unique lock definition used in [24] and [23] stems from our handling of `Thread.join()`, described in Section 2.1.4. In `mtrt`, two child threads access I/O statistics while holding a common lock `syncObject`, and then a parent thread accesses the same statistics after calling `join()` on each of the child threads. If S_1 and S_2 were the pseudolocks we introduced for the child threads, then the locksets held while accessing the statistics variables were $\{S_1, \texttt{syncObject}\}$, $\{S_2, \texttt{syncObject}\}$, and $\{S_1, S_2\}$. Since no unique lock protects the shared variables, these accesses are incorrectly identified as a datarace under the unique lock datarace definition, while the common lock definition correctly identifies the accesses as safe.

The necessity of handling thread creation orderings is illustrated by the “NoOwnership” column of Table 6.2. Most of our test programs use a parent thread to initialize data which is then passed to a child thread without synchronization, which leads to many false datarace reports without our ownership model. Our handling of `join()` with pseudolocks has a less dramatic effect on accuracy, probably because `join()` is more rarely used in Java. However, for `sor2`, which uses `join()` to safely

Example	Base	Full	NoStatic	NoDominators	NoCache
`mtrt`	9.0s	10.9s (20%)	Out of Memory	10.9s (21%)	11.4s (26%)
`tsp`	10.0s	14.2s (42%)	27.5s (175%)	15.7s (57%)	382s (3722%)
`sor2`	2.4s	2.7s (13%)	2.7s (13%)	9.8s (316%)	3.2s (37%)

Table 6.3: Runtime Performance

read many values from an array, another 1000 false dataraces are reported if our pseudolock technique is not employed.

6.3 Performance

The runtime performance of our algorithm with different sets of optimizations enabled is detailed in Table 6.3. The "Base" column denotes performance with no datarace detection performed, and the "Full" column gives the runtime for performing datarace detection with all optimizations enabled. The other columns reflect runs with a single optimization disabled. "NoStatic" disables the static datarace analysis described in Section 5.1. The instrumentation elimination and loop peeling passes described in Section 5.2 are turned off for the "NoDominators" column. Finally, the dynamic cache described in Chapter 4 is disabled for the "NoCache" column.

Our performance numbers were obtained as follows. Each test was run five times in one execution of the Jalapeño virtual machine, and the best runtime is reported (thereby negating compilation costs). Jalapeño's full optimizing compiler was utilized, but no adaptive compilation was performed. We used a 1GB heap to help ensure that no garbage collection occurred during the executions. The machine used for the tests ran AIX and had a single 450MHz POWER3 CPU.

As seen in Table 6.3 and Figure 6-1, our overheads with all optimizations enabled are quite low, less than those reported for any previous dynamic datarace detection system. No single optimization seems to obviously be the most effective, as different optimizations dramatically affect performance on different benchmarks. `mtrt` contains a large number of accesses which can never participate in dataraces, making our

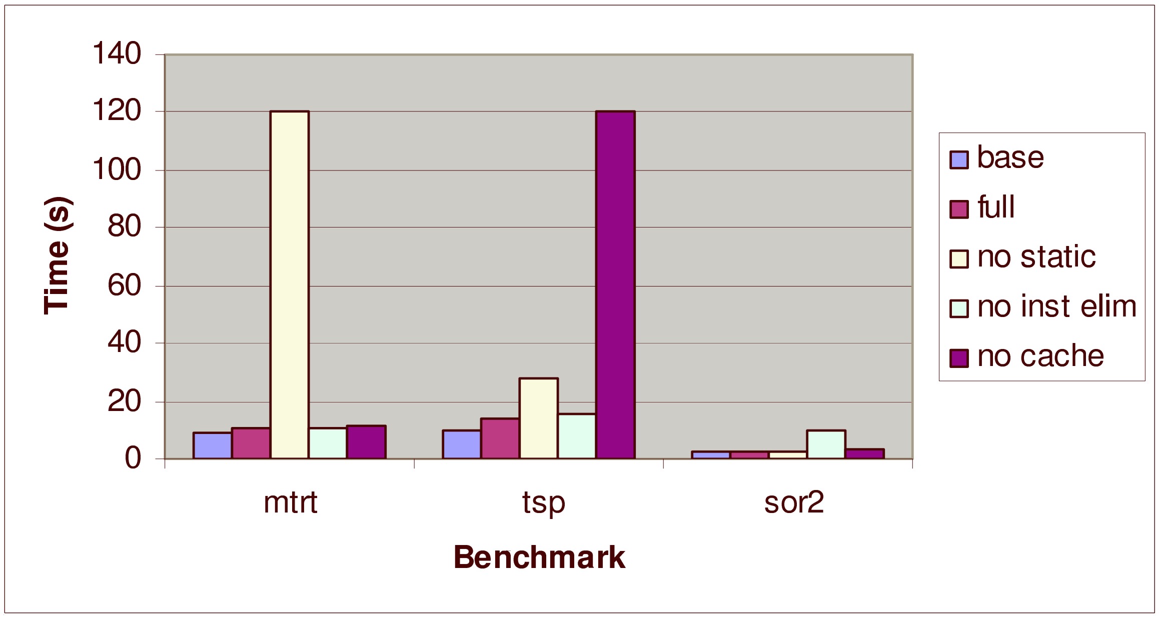


Figure 6-1: A bar chart view of the performance results. The "no inst elim" bars correspond to the "NoDominators" column in Table 6.3.

static datarace analysis critical (without the analysis, the program generates so many access events that we run out of memory before termination). Programs with many loops over arrays, such as `sor2`, benefit greatly from loop peeling and instrumentation elimination. Finally, our dynamic cache becomes important when our other optimizations are less effective, such as for `tsp` which has many recursive method calls and loops containing method calls that make static analysis more difficult.

Our use of Jalapeño makes measuring space overhead difficult, as Jalapeño mixes program and virtual machine data in one heap. Our worst memory overhead was for the `tsp` benchmark, with approximately 500K of memory used by our instrumentatation. This memory overhead includes about 16K of memory per thread for our dynamic caches, plus storage of 7967 trie nodes which held history for 6562 memory locations. Our trie packing scheme greatly decreases memory overhead in some cases, with almost all tries representing the maximum of 32 possible memory locations.

Chapter 7

Related Work

As indicated in Chapter 1, there is a large body of past work on datarace detection. Much of the earlier work targeted programs that utilized a *fork-join* parallelism model. The fork-join model for multithreading allows for simpler reasoning about which computations can be executing in parallel, since each forked child thread must be joined at some later point. In Java, `join()` need not be invoked on a child thread, so in many cases analyses must conservatively assume that the child thread may run in parallel with any code of the parent thread executing after the child thread is started. Because of the differing models of parallelism, a large portion of previous work is not directly applicable to our target object-oriented applications. Also, much of this past work defined dataraces based on the happened-before relation [18], which we do not employ in our work. For an excellent summary of much of the work on datarace detection for fork-join programs, see [15].

Some of Edith Schonberg's work on dynamic datarace detection contains ideas similar to the ones we employed in our system [25, 13]. In [13], the idea of a *lock cover* is introduced, corresponding to what we call a lockset, and the advantages of using lock covers in addition to the happened-before relation for defining dataraces is discussed. Also, an optimization called *subtraction* which has many similarities to our notion of *weaker-than* is described. However, subtraction is only described as an optimization to the datarace detection algorithm itself, while we use weaker-than in several other stages of our race detection system to improve performance. Also, no implementation details or overhead results are given in Schonberg's work.

Eraser was the first dynamic datarace detection system to use a purely lock-based approach and target object-oriented programs [24]. As discussed in Chapter 2, one difference between our system and Eraser is that we detect common lock dataraces, while Eraser detects unique lock dataraces. In practice, this only seems to make a difference because of our handling of `join`, which is not handled by Eraser (see Chapter 6 for details). Our ownership model is based on the model employed in Eraser. Eraser works independently of the source language of the program by instrumenting compiled binaries, while our current implementation is only for Java programs. The runtime overhead of Eraser is quite large, from 10× to 30×, and [24] states that low overhead was not an important goal of their system.

The *object race detection* system of Praun and Gross uses several techniques to greatly decrease its overhead [23]. Like our system, Praun and Gross use a static escape analysis to detect statements which cannot be involved in dataraces and filter them from consideration. Then, their system dynamically detects dataraces at the object level instead of at the granularity of individual memory locations. These optimizations lead to an overhead of 16% to 129% for the same benchmarks that we used, with less than 25% space overhead. But, as discussed in Chapter 6, the coarseness of the object race definition leads to the reporting of many false positives, which we feel greatly hampers the usefulness of any debugging tool. Object race detection also uses an owernship model similar to Eraser's for handling thread initialization. Finally, our static analysis goes beyond escape analysis to simulate dynamic datarace conditions statically, in some cases allowing us to make our set of potentially racing instructions smaller than what would be found with just escape analysis.

Christiaens's TRaDe system differs from other recent systems in its use of the happened-before relation for detecting dataraces [12]. TRaDe's overhead is higher than ours, approximately 4× to 15× over an interpreter with about 3× space overhead. Also, TRaDe does no static analysis, but instead performs escape analysis dynamically to decrease overhead. Two recent commercial products which provide datarace detection systems are AssureJ [17] and JProbe [16]. Unfortunately, few technical details could be discovered about these systems. The time overhead for AssureJ has been measured as 3× to 30×, while JProbe's large memory requirements make it impractical for use on programs of reasonable size [12].

Static datarace detection for Java has been an active area of recent research. Static methods are appealing because they can soundly ensure that a program is free of dataraces, while dynamic approaches cannot. Our static datarace analysis requires no annotations and is based on escape and points-to analysis [10]. Other static detection schemes for Java require either annotations or the use of alternate constructs for shared data. Flanagan and Freund use type-based equivalence of lock variables with annotations in their static tool for Java [14]. Bacon's Guava is a dialect of Java which disallows dataraces [4]. Guava forces all shared objects to be instances of the *Monitor* class category, and proper synchronization to these Monitor objects is enforced statically, thereby eliminating the possibility of dataraces. Boyapati and Rinard have developed a system of type annotations for Java which ensure that a well-typed program is datarace-free [5]. Their system is also flexible in that it allows for a generic class to be subclassed with different protection mechanisms, so for example the thread-safe `Vector` class and the unsafe `ArrayList` class from Java's class libraries could both be derived from the same base class. Past work on static datarace analysis for languages besides Java include Warlock [26], an annotation-based tool for C which supports lock-based synchronization, and Aiken and Gay's system for detecting dataraces in SPMD programs [1].

Chapter 8

Conclusions and Future Work

We have presented a novel approach to dynamic datarace detection for object-oriented programs. Our approach is precise, with almost all reported dataraces corresponding to actual bugs in our test programs. We also employ complementary static and dynamic optimization techniques to greatly decrease the overhead of our system to between 13% and 42% for our test cases, well below previous work. We believe that our techniques could be used as the basis for performing datarace detection in a production system.

Many interesting issues remain for future work. One drawback of our approach is its use of whole program static analysis to decrease overhead. This analysis may not scale well for large programs, and it requires that the whole program be presented to it as input, which may not be available in some cases. In the future, we hope to investigate methods for converting this whole program static analysis to an analysis performed at JIT compilation time, perhaps for only parts of the program. Ideally, we would be able to get many of the benefits of the whole program analysis without too much extra runtime overhead. We also think that in general our approach of applying both static and dynamic analysis could be useful for other problems such as deadlock detection and immutability analysis.

Also, we are working on a new infrastructure for integrating our datarace detection techniques with the record/replay techniques of DejaVu [7] and applying these techniques to a broader range of programs. This platform could provide many new debugging techniques, such as taking a detected datarace and creating a malicious thread schedule that illustrates the dangerous nature of the bug. We hope to integrate our various bug detection techniques into a powerful platform for reasoning about the behavior of multithreaded programs.

Bibliography

[1] Alexander Aiken and David Gay. Barrier inference. In *Proceedings of the 25th Symposium on Principles of Programming Languages (POPL)*, pages 342–354, January 1998.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[3] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, 1988.

[4] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of Java without data races. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2000.

[5] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.

[6] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. *Pro-*

ceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, 1998.

[7] Jong-Deok Choi, Bowen Alpern, Ton Ngo, Manu Sridharan, and John Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th IEEE International Parallel & Distributed Processing Symposium*, April 2001.

[8] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19, 1999.

[9] 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 *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, June 2002.

[10] Jong-Deok Choi, Alexey Loginov, and Vivek Sarkar. Static datarace analysis for multithreaded object-oriented programs. Report RC22146, IBM Research, 2001.

[11] Jong-Deok Choi and Sang Lyul Min. Race frontier: Reproducing data races in parallel-program debugging. In *Proceedings of Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.

[12] Mark Christiaens and Koen De Bosschere. TRaDe, a topological approach to on-the-fly race detection in Java programs. *Proceedings of the Java Virtual Machine Rsearch and Technology Symposium (JVM'01)*, April 2001.

[13] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):85–96, 1991.

[14] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232, June 2000.

[15] D.P Helmbold and C.E. McDowell. A taxonomy of race detection algorithms. Technical Report UCSC-CRL-94-35, University of California, Santa Cruz, 1994.

[16] KL Group, 260 King Street East, Toronto, Ontario, Canada. Getting Started with JProbe.

[17] Kuck & Associates, Inc., 1906 Fox Drive, Champaign, IL 61820-7345, USA. AssureJ User's Manual, 2.0 Edition, March 1999.

[18] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[19] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *Programming Languages and Systems*, 1(1):121–141, 1979.

[20] Jeremy Manson and William Pugh. Semantics of multithreaded Java. December 2001.

[21] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 135–144, June 1988.

[22] Robert H. B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, 1992.

[23] Christoph von Praun and Thomas Gross. Object race detection. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.

[24] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[25] Edith Schonberg. On-The-Fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 285–297, June 1989.

[26] Nicholas Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[27] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.