Optimized Run-Time Race Detection And Atomicity Checking Using Partial Discovered Types *

Rahul Agarwal Amit Sasturkar Liqiang Wang Scott D. Stoller

ABSTRACT

Concurrent programs are notorious for containing errors that are difficult to reproduce and diagnose. Two common kinds of concurrency errors are data races and atomicity violations (informally, atomicity means that executing methods concurrently is equivalent to executing them serially). Several static and dynamic (run-time) analysis techniques exist to detect potential races and atomicity violations. Run-time checking may miss errors in unexecuted code and incurs significant run-time overhead. On the other hand, run-time checking generally produces fewer false alarms than static analysis; this is a significant practical advantage, since diagnosing all of the warnings from static analysis of large codebases may be prohibitively expensive.

This paper explores the use of static analysis to significantly decrease the overhead of run-time checking. Our approach is based on a type system for analyzing data races and atomicity. A type discovery algorithm is used to obtain types for as much of the program as possible (complete type inference for this type system is NP-hard, and parts of the program might be untypable). Warnings from the type-checker are used to identify parts of the program from which run-time checking can safely be omitted. The approach is completely automatic, scalable to very large programs, and significantly reduces the overhead of run-time checking for data races and atomicity violations.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software/Program Verification**]: Formal Methods, Reliability; D.2.5 [**Testing and Debugging**]: Debugging Aids

Keywords

Atomicity, Data Races, Type System, Performance

General Terms

Languages, Verification, Performance, Reliability

*Address: Computer Science Dept., SUNY at Stony Brook, Stony Brook, NY 11794-4400. This work was supported in part by NSF under Grants CNS-0509230 and CCR-0205376 and by ONR under Grants N00014-02-1-0363 and N00014-04-1-0722. Email: {ragarwal,amits,liqiang,stoller}@cs.sunysb.edu Web: http://www.cs.sunysb.edu/~{ragarwal,amits,liqiang,stoller}

ASE'05, November 7–11, 2005, Long Beach, California, USA.

1. INTRODUCTION

Concurrent programs are notorious for containing errors that are difficult to reproduce and diagnose at run-time. Some common kind of programming errors include data races and atomicity violations. A *data race* occurs when two threads concurrently access a shared variable and at least one of the accesses is a write. Atomicity is a common higher-level correctness requirement that expresses non-interference between concurrently executed methods. A method is *atomic* if every execution of the program is equivalent to an execution in which the atomic method is executed without being interleaved with other concurrently executed methods.

Several tools have been developed for run-time detection of data races [19, 20, 7, 17] and atomicity violations [11, 21] in concurrent programs. The run-time race detection algorithms check that a locking discipline is followed to ensure absence of races, *i.e.*, every variable shared between threads is protected by a mutual exclusion lock. The run-time atomicity checking algorithms check whether locks are used in a way that ensures atomicity. Run-time checking cannot guarantee absence of data races or atomicity violations in other executions of the program. Run-time checking also incurs a significant run-time overhead, slowing down programs by a factor of 1.5 to 75. On the other hand, run-time checking generally produces fewer false alarms than static analysis; this is a significant practical advantage, since diagnosing all of the warnings from static analysis of large codebases may be prohibitively expensive.

Another approach is static analysis, in particular, type systems, that can statically ensure race-freedom and atomicity [6, 9, 12, 18]. However, manually annotating code with the necessary type annotations can be a significant burden, especially for legacy code.

Type inference reduces the annotation burden by automatically determining types for some or all parts of the program. A type inference algorithm is *complete* if it can infer types for all typable programs. Unfortunately, complete type inference is NP-complete for these type systems. This

motivated the development of incomplete type inference algorithms. *Type discovery* is an inexpensive approach to type inference that employs both run-time monitoring and static analysis to infer types for all or part of a program. Type discovery is incomplete but experience shows it is very effective in practice, discovering 98% of the annotations in our experiments [1]. However, static approaches generally produce more false alarms than runtime techniques. Furthermore, although type discovery is nearly linear-time and in that sense very scalable, because it is guided by run-time monitoring, it generally cannot discover types for methods not exercised by the test suite (it generally can discover types for unexecuted branches in executed methods); for large codebases without good test suites, this may be a significant fraction of the code. [1]

This paper explores the use of static analysis to significantly decrease the overhead of run-time checking. First, type discovery is used to discover types for all or part of the program. The discovered types are then given to the typechecker, which issues warnings. Run-time race detection and atomicity checking is then focused on fragments of code for which the type checker issued warnings. The paper presents three approaches to focus run-time race detection, the first based on instrumenting (potentially) race unsafe field accesses, the second based on instrumenting untyped methods, and the third based on combining the above two techniques. It also shows how to use atomicity types to reduce the overhead of an accurate run-time atomicity checking algorithm. Our experiments with over 175 KLOC show that these techniques can significantly reduce the runtime overhead. Also, the run-time race detection produces significantly fewer spurious warnings than the typechecker and can provide more detailed and specific diagnostic information. The user can manually inspect the run-time warnings, which are more likely to indicate real errors, and then, if desired, the typechecker warnings. The user can then use the detailed mode of [17] to further reduce the warnings. The approach is completely automatic, scalable to very large programs, and significantly reduces the overhead of run-time checking for data races and atomicity violations. Although type discovery requires running an instrumented program, the cost is much less than full run-time checking, because sampling (*e.g.*, monitoring a few instances of each class) suffices for type discovery; furthermore, the cost of type discovery is amortized across all subsequent testing in which the discovered types are used to reduce the overhead of run-time checking.

We are also exploring static type inference for type systems that ensure absence of deadlock [5] and ways to use those types to reduce the overhead of run-time detection of potential deadlocks [13]. Our ultimate goal is to reduce the overhead of run-time checking to a level where it can be used unobtrusively throughout the testing process, or even in deployed systems, instead of only during a limited period of testing focused on concurrency errors.

Rest of the paper is organized as follows. Section 2, 3, 4 and 5 describe run-time race detection and atomicity checking, type systems for race-freedom and atomicity, type discovery for race-free types, and type inference for atomicity types respectively. Section 6 presents our techniques for focused run-time checking. Section 7 briefly describes the implementation. Sections 8 and 9 present an experimental evaluation of the techniques. Section 10 presents related work.

[1]SAT-based type inference for these type systems [10] don't have this limitation but, because they are directly attacking an NP-hard problem, probably don't scale beyond ∼50 -100 KLOC.

2. RUN-TIME RACE DETECTION AND ATOMICITY CHECKING

2.1 Run-time race detection

We use the popular lockset algorithm (sometimes called the Eraser algorithm [19]). The basic idea is that violations of a simple locking discipline—in which a shared variable is consistently protected by the same lock—indicate potential data races.

The lockset algorithm maintains a set $C(v)$ of candidate locks for each shared variable v. The set contains locks that have protected v during the computation so far. During initialization of v, the set $C(v)$ contains all locks. On each subsequent access to v, $C(v)$ is updated to be the intersection of $C(v)$ and the set of locks held by the thread. If $C(v)$ ever becomes empty, it indicates violation of the simple locking discipline, and a warning of a possible race on v is issued. If v is never updated after it becomes shared (*i.e.*, after a second thread has accessed it), v is called "readonly" and no warning is reported for it, even if its lockset becomes empty.

Our implementation of the Eraser algorithm in Java maintains the following attributes per object: *lkset*: set of locks held whenever any field of the object is accessed; *rdonly*: a boolean that is true if none of the fields of the object are updated after initialization; and *shar*: a boolean which indicates whether the object is shared (*i.e.*, has been accessed by multiple threads). When an object o is first allocated, the attribute *rdonly* is true and attribute *shar* is false for o. The *lkset* contains all possible locks. The attributes of o are not updated until a second thread accesses a field of o (this is interpreted as indicating that initialization is finished), whereafter *shar* attribute is set to true, *lkset* is updated with the intersection of the current *lkset* and the set of locks held by the current thread, and *rdonly* is set to false when a field of o is written. A race is reported on object o if *lkset* is empty, *shar* is true, and *rdonly* is false for o.

Although we use the Eraser algorithm at object granularity (*i.e.*, one *lkset*, *shar*, and *rdonly* attributes are maintained for each object) in the experiments in Section 8, our approach can equally well be used with any variant of the Eraser algorithm, such as [19], at object granularity or field granularity (*i.e.*, one *lkset*, *shar*, and *rdonly* attribute is maintained for each field of each object).

2.2 Run-time atomicity checking

We developed two runtime algorithms for detecting potential violations of atomicity: reduction-based algorithm (independently developed by [11]) and block-based algorithm [21]. Both algorithms do not simply look for actual violations of atomicity in the observed execution, but also check whether a violation seems possible in other executions, based on how synchronization is used in the observed execution.

The reduction-based algorithm uses an enhanced lockset algorithm to determine whether there is a data race on each

variable and then uses this information to determine commutativity of events. If the sequence of events in each transaction matches a given commutativity pattern based on Lipton's reduction [16], then the transactions are atomic. A transaction is an execution of a code block that is expected to be atomic. Ideally, the programmer would indicate those code blocks. In our experiments, we rely on simple heuristics to identify transactions; roughly, each execution of a public method or synchronized block is a transaction. Synchronized blocks are included because locks are often used to achieve atomicity. We consider public methods as transactions because they are abstractions often expected by clients of the class to behave as atomic operations.

The block-based algorithm determines whether a violation of atomicity is possible in traces (*i.e.*, sequences of events) obtained from the observed execution by permuting the order of events consistent with the synchronization events. Explicitly computing these permutations would be prohibitively expensive. The block-based algorithm looks for such violations by constructing "blocks"—a block is an abstraction that captures the information about two events that is relevant to atomicity checking–and then comparing each block with all blocks in other transactions. If two blocks are found whose events can be interleaved in a way that matches a fixed set of unserializable patterns, the transactions containing them are not atomic. The unserializable patterns are defined based on the concept of view serializability in database transaction processing. For example, one unserializable pattern is when a write in one transaction can occur between two consecutive reads in another transaction. The block-based algorithm is more expensive than the reduction-based algorithm, but is more accurate, *i.e.*, reports fewer false alarms. This paper focuses on optimizing the block-based algorithm, which we now sketch in more detail.

There are two kinds of blocks: 1v-block and 2v-block. *1v-block* consists of two events that access the same variable in a single transaction. For an event e that accesses variable x in a transaction t, 1v-blocks are constructed as follows: if there is a write event to x in t that precedes e, a 1v-block is created from that write and e; otherwise, if there is a read of x in t that precedes e, a 1v-block is created from the read and e; in addition if e is the final write to x in t, 1v-blocks are created from e and every initial read of x in t (*i.e.*, they precede the first write of x in t). A *2v-block* is created from two accesses to different variables in the same transaction, and the two events are initial reads or final writes (*i.e.*, writes not followed in the same transaction by another write to the same variable). Each block records the variable(s) accessed, the operations (read or write), and synchronization information— the locks held at each operation, the locks held continuously from one event to another, etc. Note that multiple pairs of events can produce the same block.

For all pairs of transactions t_1 and t_2, every 1v-block of t_1 is analyzed with each 1v-block of t_2, and every 2v-block of t_1 is analyzed with each 2v-block of t_2. To analyze two 1v-blocks, the algorithm checks whether any three events from them can be interleaved (consistent with the recorded synchronization) to form an unserializable pattern. Analysis of two 2v-blocks is similar, except that the interleaving of all four events in them is considered.

Our implementation of the block-based algorithm incorporates several optimizations. For example, the most recent several event patterns are cached, and when an event pattern in the cache appears again, we do not construct blocks from it again. Also, the blocks of each transaction are stored in a set; this ensures that only one copy of each block is stored. Even with these optimizations, the block-based algorithm incurs relatively high overhead, with a median slowdown of 35 in the experiments in [21]. Details of the algorithm and a correctness theorem and proof appear in [21].

3. TYPE SYSTEMS FOR RACE-FREEDOM AND ATOMICITY

3.1 Type systems for race-freedom

The type systems for race-freedom [6, 18, 9] are based on the locking discipline described in Section 2.1. They require annotation of the program to indicate which lock protects each field and variable. The typing rules ensure that the appropriate lock is held when the field or variable is accessed. Soundness of the type system means that a well-typed program is race-free. We present overviews of Parameterized Race Free Java [6], of our extensions to it [18], and finally of atomicity types [12, 18].

3.1.1 Overview of Parameterized Race Free Java

In Parameterized Race Free Java (PRFJ) [6] as in its predecessor Race Free Java [9], types are extended to indicate the synchronization discipline (also called 'protection mechanism' or 'owner') used to co-ordinate accesses to each object. To allow different instances of a class to use different protection mechanisms, each class is parameterized by formal owner parameters which may be instantiated with other formal owner parameters, final expressions (*i.e.*, expressions whose value does not change) representing locks, or special owners (described below). The first owner parameter of each class indicates the owner of the `this` object; the other owner parameters are used to propagate ownership information to the object's fields.

A final expression used as an owner specifies a lock that must be held when the object is accessed. There are four special owners: `thisThread`, `self`, `readonly` and `unique`. `readonly` indicates that the object is readonly and cannot be updated. `unique` means that there is a unique reference to the object. `thisThread` means that the object is thread-local (*i.e.*, unshared). `self` means that the object is protected by its own lock (*i.e.*, a self-synchronized object). The owner of an object is said to guard all of its fields.

Method declarations may have a `requires` clause that contains a set of final expressions; the locks on the owners of these expressions must be held when the method is invoked.[2] The special owners `thisThread`, `unique` and `readonly` are always assumed to be in the lockset. PRFJ ensures that whenever a field of an object is accessed, either the object is readonly, or the accessing thread either has a unique reference to the object or holds the lock on the root owner of the object, thus avoiding races.

PRFJ, like Java, is not not purely statically typed and allows downcasts whose safety is checked at run-time. In PFRJ, downcasts to types with multiple owner parameters require checking at run-time that the owner parameters other than the first have appropriate values [4].

[2] For simplicity, we ignore the distinction between owners and root owners in this overview.

```
class Node<thisOwner> {
  int value;
  void setValue(int v) requires this {
    value = v;
  }
  void getValue() {
    return value;
  }
}

Node<thisThread> n1 = new Node<thisThread>();
Node<self> n2 = new Node<self>();
n1.setValue(10);
fork(n2) {n2.setValue(20)};
fork(n2) {synchronized (n2) { n2.setValue(30);}}
```

Figure 1: An example program in PRFJ

The PRFJ program in Figure 1 illustrates how the PRFJ type system discovers potential race conditions. The program defines a `Node` class with a single formal owner parameter, `thisOwner`. The write to `this.value` in `setValue` is well-typed because the `requires this` clause on `setValue` implies that the lock on the owner of `this` should be held at all call sites for `setValue`. However, the read of `this.value` in `getValue` generates a type error, and indeed can be involved in a race, because the lock on the owner of `this` might not be held there.

Since `n1` is parameterized by `thisThread`, it is a thread-local object; accesses to fields of `n1` cannot be involved in any data races. The method invocation `n1.setValue(10)` typechecks because the root owner of `n1` is `thisThread`, which is always assumed to be held. Since `n2` is parameterized by `self`, it is a self-synchronized object that can be shared by multiple threads. The method invocation `n2.setValue(20)`, which writes to the field `value`, can be involved in a data race, since no lock is held when this method is invoked. PRFJ detects this race condition by checking that the `requires` clause on `setValue` is satisfied at all its invocation sites. Since the owner of `n2` is `n2` itself, and the lock on `n2` is not held when `n2.setValue(20)` is called, this invocation of `setValue` does not typecheck. But, `n2.setValue(30)` typechecks, since it is enclosed within a `synchronized(n2) {..}` block.

3.2 Atomicity types

Flanagan and Qadeer's type system for atomicity [12] extends a race-free type system [9] to associate an atomicity with each expression and statement (for brevity, "expression" means "expression or statement" in the rest of this section). The atomicity of each method is declared in the program; atomicities of other expressions are implicit. An atomicity is a *basic atomicity* or a *conditional atomicity*. The basic atomicities and their meanings are: `const`: evaluation of the expression does not depend on or change any mutable state; `mover`: the expression left-commutes with every operation of another thread that could occur immediately before it and right-commutes with every operation of another thread that could occur immediately after it (*i.e.*, the two operations can be swapped, and this still leads to the same state); `atomic`: evaluation of the expression is always equivalent to evaluation of the expression without interleaved actions of other threads; `cmpd` (compound): none of the preceding atomicities apply; `error`: evaluation of the expression violates the locking discipline specified by the race-free types.

Conditional atomicities are used when the atomicity of an expression depends on which locks are held by the thread evaluating it. A conditional atomicity $l\,?\,a : b$ is equivalent to atomicity a if lock l is held when the expression is evaluated, and is equivalent to atomicity b otherwise. $l\,?\,a$ abbreviates $l\,?\,a$: `error`.

Let α and a, b range over basic atomicities and atomicities, respectively. Each atomicity a is interpreted as a function $[\![a]\!]$ from the set ls of locks currently held to a basic atomicity: $[\![\alpha]\!](ls) = \alpha$ and $[\![l\,?\,a_1 : a_2]\!](ls) =$ if $l \in ls$ then $[\![a_1]\!](ls)$ else $[\![a_2]\!](ls)$. A partial order $\sqsubseteq$ on atomicities is defined. The ordering on basic atomicities is `const` $\sqsubseteq$ `mover` $\sqsubseteq$ `atomic` $\sqsubseteq$ `cmpd` $\sqsubseteq$ `error`. The ordering on conditional atomicities is the pointwise extension of the ordering on basic atomicities, *i.e.*, $a \sqsubseteq b$ iff $\forall ls : [\![a]\!](ls) \sqsubseteq [\![b]\!](ls)$. Rules for effectively determining the ordering on atomicities appear in [12].

The typing rules express the atomicity of an expression in terms of the atomicities of its subexpressions using five operations on atomicities: sequential composition $a; b$, iterative closure a^*, join $a \sqcup b$ (based on the partial order $\sqsubseteq$ described above), conditional $l\,?\,a : b$, and the operation $S(l, a)$ described below. Sequential composition for basic atomicities is defined by: $\alpha_1; \alpha_2$ equals `cmpd` if α_1 and α_2 are both `atomic`, and equals $\alpha_1 \sqcup \alpha_2$ otherwise. The iterative closure a^* denotes the atomicity of an expression that repeatedly executes an expression with atomicity a. For basic atomicities, it is defined by: a^* equals `cmpd` if a is `atomic`, and equals a otherwise. Sequential composition and iterative closure for conditional atomicities are defined as follows [12].

$$(l\,?\,a : b)^* = l\,?\,a^* : b^*$$
$$(l\,?\,a_1 : a_2); b = l\,?(a_1; b) : (a_2; b)$$
$$\alpha; (l\,?\,b_1 : b_2) = l\,?(\alpha; b_1) : (\alpha; b_2)$$

In this paper, we use our atomicity type system Extended Parameterized Atomic Java (EPAJ) [18], which combines Flanagan and Qadeer's atomicity types with a more expressive race-free type system that we developed by extending PRFJ to allow a different owner for each field of an object [18].

4. DISCOVERY OF RACE-FREE TYPES

Our type discovery algorithm for race-free types has three main steps [1, 3]. First, the target program is instrumented by an automatic source-to-source transformation and executed on test inputs. The instrumented program monitors accesses to fields of certain objects of each class and writes a log containing relevant information: which locks were held when the object was accessed, whether multiple threads accessed the object, etc. Second, the information in the log file is used to infer owners for fields, method parameters and return values, and owners in class declarations. Third, the intra-procedural type inference algorithm in [6] is used to infer the owners in the types of local variables and in the types of allocation sites whose owners have not already been determined. Local type-inference has the crucial effect of propagating type information into branches of the program that were not exercised in the monitored executions.

Type discovery is not guaranteed to produce correct typings for all typable programs, but experience shows that it is very effective in practice. In our experiments in [1], 98% of the race-free types were automatically discovered. We used only simple test inputs that accompanied the benchmarks. These simple test inputs exercised most of the methods, but did not necessarily achieve high branch coverage. Thus, in our experience, the result of the type discovery algorithm is mainly affected by which methods are exercised by the test inputs, and is otherwise mostly insensitive to the choice of inputs. For the larger benchmarks we are now targeting, a small and simple test suite might not exercise many methods, and type discovery will be effective only for the exercised methods.

5. INFERENCE OF ATOMICITY TYPES

This section briefly describes our algorithm for inferring the atomicity of each method. [18] contains a more detailed description. The algorithm assumes that the program is at least partly annotated with race-free types (the `requires` clauses are ignored; required locks are computed as part of conditional atomicities). The algorithm does not assume that the given race-free types are correct, and it can determine atomicity of methods in programs with races. However, errors in the given race-free types generally lead to weaker conclusions about atomicity. For example, specifying an incorrect owner may cause the algorithm to infer that a method has atomicity `cmpd` or `error`, even though the method might actually be `atomic`.

For a set S and number k, let S^* denote the set of finite sequences of elements of S, and $S^k = \{s \in S^* : |s| = k\}$. For a sequence s, let $s[i]$ denote the i^{th} element of the sequence.

Let M denote the set of all methods in the program. Let $\text{Invoke} : M \rightarrow M^*$ be a function such that $\text{Invoke}(m)$ is a sequence containing the methods that m invokes; the order of elements in this sequence is arbitrary but fixed (*e.g.*, lexicographic order). Let *Atom* denote the set of atomicities.

For each method m, the definition of m and the EPAJ typing rules [18] define an atomicity transfer function f_m, with type $f_m : Atom^{|\text{Invoke}(m)|} \rightarrow Atom$. The arguments to f_m are the atomicities of the methods that m invokes. f_m returns the atomicity of method m. Basically, $f_m(a_1, a_2, \ldots, a_n)$ applies the atomicity typing rules to the body of m in a bottom-up manner, using a_i to determine the atomicity of a call to a method $\text{Invoke}(m)[i]$, and thereby constructs the atomicity of m.

The type inference algorithm determines an atomicity for each method. This is expressed as an atomicity assignment $\text{atom} : M \rightarrow Atom$. An atomicity assignment is *consistent* if, for all methods m, the atomicity assigned to m is greater than or equal to the atomicity computed for m using the transfer function f_m, *i.e.*,

$$f_m(\text{atom}(\text{Invoke}(m)[1]), \text{atom}(\text{Invoke}(m)[2]), \ldots) \sqsubseteq \text{atom}(m)$$

The partial order on atomicity assignments is the pointwise extension of the partial order on atomicities. Smaller atomicity assignments provide stronger guarantees, thus our aim is to compute the least consistent atomicity assignment for the program. This can be done using a simple fixed-point calculation, since all atomicity transfer functions are monotonic (because the five operations listed in Section 3.2 are monotonic) and continuous [18].

6. FOCUSED RUN-TIME CHECKING

6.1 Focused Run-time Race Detection

As discussed in Section 1, both run-time checking and type-based static checking of races have limitations, and we are exploring how to combine the two approaches to overcome some of them.

This section presents a technique for utilizing the warnings from the type checker to identify parts of the program from which run-time checking can safely be omitted; in other words, we focus run-time checking on parts of the code. We present three techniques for this: *field-based*, *method-based* and *combined-field-and-method based*.

6.1.1 Field-based focused runtime checking

Given a program annotated (in whole or part) with (possibly incorrect) PRFJ types, our PRFJ type-checker can be used to compute a list of fields that might potentially be involved in data races. We refer to these as *race-unsafe* fields. The other fields are guaranteed to be race-free, *i.e.* accesses to them cannot be involved in data races. Thus, during run-time race detection, there is no need to monitor accesses to race-free fields. Field-based focused runtime checking runs the Eraser algorithm only for accesses to race-unsafe fields.

Computing race-unsafe fields automatically is essential to automate this approach. Due to the presence of `requires` clauses, it is not sufficient to consider just the fields for which the type-checker reports potential data races as candidates for race-unsafe fields. For example, consider the program in Figure 1. Due to the `requires this` clause on the method `setValue`, the type-checker does not report a race on field `value`. But `value` is a race-unsafe field, because the call `n2.setValue(20)` occurs without the lock on `n2` being held, leading to a potential data race on `n2.value`.

Our algorithm for computing race-unsafe fields works as follows. For a method m, let $requires(m)$ denote its `requires` clause. For each method m, we compute the set of locks in its `requires` clause held at all invocations of method m in all the executions. We call this set *locks-def-held*(m).

For each method m, *locks-def-held*(m) is computed as follows. Given a program P, let $Invoke = (V, E)$ denote the extended method invocation graph of P, where the vertex set V consists of all methods and all call-sites of methods and the edge set E consists of edges from each call site to the possibly called methods (there might be more than one, due to method overriding). For a call-site c, let $parent(c)$ denote the method containing c, let $called(c)$ denote the called method with the most general type, and let *locks-acquired*(c) denote the set of locks acquired in $parent(c)$ and held at c. For languages like Java that have block-structured statements for acquiring and releasing locks, *locks-acquired* is well-defined and can be computed easily with a linear scan of every method.

As described in Section 3.1.1, for a method m, the locks in $requires(m)$ are final expressions that may be method parameters of m, or the final expression `this`. The PRFJ typing rules [6] specify how these expressions are instantiated at call-sites of m. Let $instantiate(c, l)$ be a function that takes a call-site c and a lock $l \in requires(called(c))$ as arguments, instantiates l according to the typing rules, and computes and returns the root owner of l.

The function *locks-def-held* is the greatest fix-point solution to the following set of constraints. The fix-point is

computed by starting with $in(m) = requires(m)$ for all m, and iterating until a fix-point is reached.

$in(c) = in(parent(c)) \cup locks\text{-}acquired(c)$
$out(c) = \{\ l \mid l \in requires(called(c)) \wedge instantiate(c, l) \in in(c)\ \}$
$in(m) = requires(m) \cap \bigcap_{(c,m)\in E} out(c)$
$locks\text{-}def\text{-}held(m) = in(m)$

The above analysis is sound as a whole program analysis, i.e., given a well-formed whole program whose only entry points are `main` and `run` methods, and these methods have empty `requires` clauses, the greatest fixed point computed above does not include spurious locks for methods reachable from the entry points; it may contain spurious locks for unreachable methods, but this does not violate our soundness requirement.

A field f in a class C is labeled race-unsafe if either (1) C is a race-unsafe class (defined below), or (2) there is an expression o with type C and a field access $o.f$ in a method m such that the rootowner of o is not in $locks\text{-}def\text{-}held(m)$.

A class C is labelled race-unsafe if the type-checker signals any of the following warnings: (1) C is not a well-formed type as defined by the rules in [6], (2) an instance of C with owner `thisThread` may escape to another thread, (3) there is a write to a `readonly` instance of C, (4) a `unique` instance of C violates the semantics of `unique`, *i.e.*, there isn't a unique reference to it, (5) the type C is involved in an assignment type mismatch (i.e., there is an assignment $e_1 = e_2$ where both e_1 and e_2 have type C, but there is a mismatch between the owners of e_1 and e_2), or (6) the type C is involved in an invocation type mismatch (i.e., there is a method invocation where the actual and formal parameters have type C, but there is a mismatch in their owner parameters).

Using type-discovery and then this algorithm, race-unsafe fields can be computed automatically.

6.1.2 Method-based focused run-time checking

As discussed in Section 4, type discovery is effective only for methods exercised in the test suite during type discovery. For large systems, this might be a small fragment of the codebase. For example, a simple client that retrieves a static webpage from the W3C's Jigsaw webserver, [15] exercises code in about 100 classes out of more than 850 classes in Jigsaw. In such cases, executed methods are likely to be well-typed, i.e., there are no errors in the types discovered for them. But many methods will be "untyped", either because there is an error in the discovered types (possibly because the method is untypable), or because the methods were not exercised hence no types were discovered for them (simple default types can be tried, as in [6], but they often get one or more types wrong in each method), or because source code for them is unavailable.

The main idea behind the method-based approach is to perform run-time checking on all accesses in untyped methods and omit it from all accesses in well-typed methods. This can potentially perform better than the field-based approach, because the test suite used for type discovery typically exercises the most frequently executed methods; if these methods are well-typed, then the method-based approach succeeds in eliminating run-time checking in places where this provides the most benefit.

We say m is a typed method if the typechecker is run on m and it does not produce any warnings for m, *i.e.*, all field accesses and call sites in m are well typed. We say that all other methods are untyped. No field accesses are monitored in typed methods, but all field accesses in untyped methods are instrumented for run-time checking. For objects allocated in typed methods, only owner information is maintained. For objects allocated in untyped methods, the attributes *lkset*, *shar*, and *rdonly* (described in Section 2.1) are maintained. In addition, at each call site where a typed method m is called from an untyped method, code is inserted to check whether the arguments' owners or attributes conform to the declared owners of the corresponding parameters of m and whether the locks specified in m's `requires` clause are held; if either of these conditions does not hold, a warning (of a potential race) is issued. To ensure that field accesses in untyped methods cannot lead to races, these accesses are checked as follows: if the accessed object was allocated in a typed method, the current access is checked for conformance with the object's owner (*i.e.*, if the owner is `thisThread`, the current thread must be the thread that allocated the object; if it is `readonly`, the access must be a read; if it is an object, the lock on that object must be held); if the object was allocated in an untyped method, the Eraser algorithm in Section 2.1 is used to update its attributes and determine whether to issue a warning.

It is not difficult to show that any race detected by full checking with Eraser is also detected by this focused checking. Furthermore, this focused checking may issue fewer false alarms than full Eraser. In particular, the type system and type discovery can recognize that objects are race-free because there are appropriately protected unique references to them. Focused checking will not monitor accesses to those objects. Eraser does not recognize unique references, so full checking with Eraser will produce false alarms on accesses to those objects.

6.1.3 Combined field-and-method-based focused run-time checking

The combined approach is based on the observation that not all field accesses in untyped methods need to be monitored. Specifically, only race-unsafe fields need to be monitored in untyped methods. Furthermore, at call sites to typed methods in untyped methods, checking whether arguments' owners or attributes conform to the declared owners of the corresponding method parameters is necessary only for parameters whose types are race-unsafe classes.

6.2 Focused Run-time Atomicity Checking

Both of the run-time atomicity checking algorithms sketched in Section 2.2 can be optimized using the race-free types and atomicity types in EPAJ, sketched in Section 3. In [18], we showed how to optimize the reduction-based algorithm using EPAJ types. In the original reduction-based algorithm, every shared variable is monitored for races, and every transaction is checked for atomicity. In the focused algorithm, we monitor only race-unsafe fields and check atomicity only for methods that are not verified to be atomic by the typechecker. In our experiments [18], this optimization reduced the median slowdown from 38.1 to 1.5.

This section presents an approach to focus the block-based algorithm [21] using atomicity types automatically produced by type discovery and type inference. The EPAJ typechecker lists all methods that it has verified to be atomic. The list contains well-typed methods with a basic atomicity less than or equal to `atomic` and methods with a conditional

atomicty that simplifies at all call sites to a basic atomicity less than or equal to `atomic`. We refer to transactions that correspond to executions of these methods as atomic transactions. We refer to the other methods as possibly non-atomic methods, and to transactions corresponding to executions of them as possibly non-atomic transactions. We optimize the block-based algorithm by reducing the number of blocks constructed using events in executions of those methods. Note that the accesses in those methods cannot be completely ignored because they may participate in forming unserializable patterns with events from other methods.

For an atomic transaction t, the focused block-based algorithm does not directly construct blocks from pairs of events in t. Instead, it records a few items characterizing the accesses performed by t, and uses that information as described below. For each escaped variable x accessed in t (a dynamic escape analysis is used to identify the escaped variables [21]), it records whether t writes (and possibly reads) x or merely reads x. It also records the set $held(t)$ of locks held at any point during t (i.e., the set of locks held when t starts or acquired at any time during t). This information is stored for all atomic transactions in a global table. The focused block-based algorithm treats each atomic transaction as if it consisted of the following events: for each escaped variable accessed by t, a single write or read (depending on whether t wrote x) event protected by the locks in $held(t)$. These reduced (i.e., relatively small) sets of events for the atomic transactions are used in the following three ways.

First, for each 1v-block from a possibly non-atomic transaction, the focused algorithm checks whether an unserializable pattern can be formed from the two events represented by the 1v-block and one event from the reduced set of events of an atomic transaction. Specifically, information about the locks held is used to determine whether the latter event can occur between the former two, and if so, whether the resulting pattern of three reads and writes matches an unserializable pattern, i.e., the middle event does not commute with the first and last events.

Second, for a possibly non-atomic transaction t that contains calls to atomic methods, the focused block-based algorithm treats those method calls as if they consisted of the reduced sets of events described above, i.e., 1v-blocks and 2v-blocks for t are constructed from the events in those sets and the other events in t, except that 1v-blocks are not constructed from two events from the same call to an atomic method. Note that the original block-based algorithm processes events in method calls in a transaction in exactly the same way as events in the top-level method call in the transaction; the block-based algorithm does not incorporate a concept of nested transactions.

Third, 2v-blocks are constructed from the reduced set of events for each atomic transaction. Thus, the focused block-based algorithm may construct 2v-blocks from two events from the same atomic transaction, but it never constructs 1v-blocks from two events from the same atomic transaction.

A proof that treating $held(t)$ as the set of locks held at each access in an atomic transaction t does not affect the result of the block-based algorithm appears in [2]. Due to space limitations, here we just sketch the two main conditions used in the proof. First, the proof relies on the structure of the EPAJ type system, specifically, the fact that a transaction statically verified as atomic does not acquire any lock after releasing a lock. Second, it relies on the assumption that there is no potential for deadlock [13, 21]. A set T of transactions has potential for deadlock if some feasible interleaving of the events in T leads to a deadlock. Potential for deadlock indicates a potential violation of atomicity, because an execution in which some transactions deadlock is not equivalent to any serial execution; a serial execution cannot deadlock, because we assume the program uses block-structured synchronization, as in Java, so each transaction releases every lock that it acquires. We use an extension of the goodlock algorithm [13] to detect potential for deadlock. Therefore, it suffices to show that our optimization is correct in the absence of potential for deadlock.

7. IMPLEMENTATION

Source-to-source transformations, implemented in the Kopi compiler (http://www.dms.at/kopi/), instrument programs for type discovery, and full and focused run-time checking. We do not instrument Java API classes. Instrumenting them would create new dependencies among the bootstrap classes; since the JVM loads those classes in a fixed order, an initialization error would occur. Since we do not perform run-time race detection for fields defined in Java API classes, we do not attempt to detect races on those fields statically either, i.e., we do not type-check Java API classes. We do check for races on all fields defined in application classes, including fields that refer to instances of Java API classes. In our benchmarks, none of the application classes require multiple owner parameters, so there is no need for run-time checking of ownership information at downcasts. Our type checker for EPAJ is based on Flanagan and Freund's Rccjava. Our implementation of atomicity type inference is based on Rccjava and uses Soot (http://www.sable.mcgill.ca/soot) to compute the Invoke function.

8. EXPERIENCE WITH RACE DETECTION

We evaluated the focused run-time race detection on seven multi-threaded programs. The first three programs (elevator, tsp and hedc) were developed at ETH Zürich and used as benchmarks in [20]. We ran tsp on two inputs, with 12 and 15 nodes. The next three programs (moldyn, raytracer and montecarlo) are part of the Java Grande Forum Benchmark Suite, available at http://www.epcc.ed.ac.uk/ . These benchmarks came with sample inputs, which we used for type discovery and to evaluate the overheads of full and focused run-time race detection.

The seventh benchmark is W3C's Jigsaw web server [15]. We wrote a simple client that retrieves a static webpage using GET requests. In our test scenario, three clients run concurrently, each makes a sequence of 1500 requests for that page. Code in only 102 classes of more than 850 classes in Jigsaw webserver code was executed. We discovered types for those 102 classes and ran the typechecker on those classes. The remaining classes are untyped hence classified as race-unsafe.

Table 1 contains experimental results. The first two columns give the name of the benchmark and the size in lines of code. The Base Time column presents the base running time (user + system time) of the benchmark. The Full Run Time column presents the running time with full run-time race detection. Columns under Field-based present the results of field-based focused checking. Rc Un Fds column presents the number of race-unsafe fields and the total number of

			Full	Field-based				Method-based				Combined		
Program	LOC	Base Time	Run Time	Rc Un Fds	Run Time	Frac Ovhd	Spdup	Untyp Meths	Run Time	Frac Ovhd	Spdup	Run Time	Frac Ovhd	Spdup
elevator	523	.61	.92	1/21	.86	41%	19%	1/24	.88	44%	13%	.89	46%	10%
tsp(12)	706	.87	3.6	12/36	1.42	63%	80%	11/24	1.99	129%	59%	1.39	60%	81%
tsp(15)	706	.82	3.29	12/36	1.53	87%	71%	11/24	1.99	143%	59%	1.57	91%	70%
hedc	7072	1.92	2.60	36/206	2.21	15%	57%	30/379	2.36	23%	35%	2.17	13%	63%
moldyn	730	34.97	1389	8/91	36.73	5%	100%	6/23	2090	5877%	-52%	35.48	1%	100%
raytracer	1308	29.96	2279	4/61	45.07	50%	99%	8/72	53.45	78%	99%	44.19	47%	99%
montecarlo	3198	33.29	142	2/94	34.02	2%	99%	4/179	34.54	4%	99%	34.17	3%	99%
Jigsaw	157K	44.4	100	3802/3854	88.75	100%	21%	6531/7650	62.36	40%	68%	60.07	35%	72%

Table 1: Experimental results with focused run-time race detection.

fields. Run Time column under Field-based presents the running time with field-based focused checking. Frac Ovhd presents the fractional overhead of the focused checking. The fractional overhead is computed as $(t_{foc} - t_{base})/t_{base}$ where t_{foc} is the running time of focused checking and t_{base} is the running time in column Base Time. Spdup column presents the speedup in the runtime checking time, *i.e.*, the fractional decrease in the overhead. Method-based present similar results for method-based focused checking and Combined columns for combined focused checking. Speedup is $(c_{full} - c_{foc})/c_{full}$ where c_{full} = cost of full race-checking = t_{full} - t_{base} and c_{foc} = cost of focused race-checking = t_{foc} - t_{base}. t_{full} is the running time with full run-time checking. Running times are measured in seconds on a 500MHz Sun Blade 100 with Sun JDK1.4 and are the average over three runs.

The field-based approach does not introduce any additional checking (compared to the full Eraser algorithm), so field-based focused checking is always at least as fast as full checking, and combined focused checking is always at least as fast as method-based focused checking. This is confirmed by our experiments; the apparent discrepancy for elevator (combined focused checking is .01 sec slower on average than method-based focused checking) is within the statistical variation (standard deviation .02 sec). The method-based introduces additional checking at the boundary between untyped and typed code, *i.e.*, at calls to typed methods from untyped methods. This can make method-based focused checking slower than full checking. This occurs in one benchmark, namely moldyn. This is because the most frequently executed methods are untyped, and they frequently call very small typed methods (*e.g.*, "get" methods). There is little correlation between the percentage of fields classified as race unsafe or methods classified as untyped and the speedups achieved. The speedups depend on how frequently the race safe fragments of the code (i.e., code with race safe fields and typed methods) are exercised.

The combined approach outperformed full checking in all benchmarks. The average and median speedups in the checking time are 74% and 77%, respectively. The average and median fractional overhead of checking with the combined approach are 37% and 40%, respectively.

The field-based approach outperformed the method-based approach in all of the smaller benchmarks, *i.e.*, all except Jigsaw. This is not surprising, since most of the methods in them are exercised by the sample inputs, and most of the fields are classified as race-free by the typechecker.

The method-based approach outperformed the field-based approach on Jigsaw, which is the largest and most interesting benchmark, with 34,839 LOC in 102 analyzed classes and 122,362 LOC in other classes. The field-based approach offered little benefit (speedup 21%), because most classes are unexercised during type discovery and hence classified as race-unsafe, and many fields accessed in the exercised classes are also exercised in the unexercised classes and hence are classified as race-unsafe, so all accesses to them are monitored with the field-based approach. The method-based performs better (speedup 68%), because it does not monitor accesses to some of those fields in well-typed methods.

9. EXPERIENCE WITH ATOMICITY CHECKING

We evaluated the focused block-based atomicity checking algorithm on the benchmarks described in the previous section except Jigsaw, because Soot failed to construct the call graph for it. The results appear in Table 2. Running times are measured in seconds on a 1GHz Sun Blade 1500 with Sun JDK 1.4 and are the average over five runs. Base Time is the running time of the original program. Intcpt Ovhd is the overhead of intercepting events, i.e., the increase in running time when all events relevant to atomicity checking (field accesses, method calls, synchronized statements, etc.) are intercepted but not processed; i.e., code is inserted to call a method with arguments describing the event, and that method simply returns. Unopt Ovhd is the cost of the unoptimized block-based algorithm, i.e., the increase in running time relative to the version that intercepts events without processing them. Foc Ovhd is the cost of the focused block-based algorithm, measured the same way. The same code is used to intercept events for the unoptimized and focused versions; our current optimization only affects the cost of processing those events. Spdup is (Unopt Ovhd - Foc Ovhd) / Unopt Ovhd. The average speedup is about 32%, i.e., the cost of the block-based algorithm is reduced by about one third.Frac Ovhd is the overhead of the optimized block-based algorithm as a fraction of the total running time, i.e., Foc Ovhd / (Base Time + Intcpt Ovhd + Foc Ovhd).

However, in some benchmarks, the overhead of intercepting events exceeds the overhead of the block-based algorithm itself, so an important direction for future work is to avoid intercepting some events. The focused block-based algorithm in Section 6.2 provides good opportunities for this. Using that algorithm, for each execution of a method classified as atomic by the type checker, for each shared variable it accesses, it is sufficient to intercept one write event or, if there is none, one read event. Thus, if static analysis can be used at instrumentation time to determine, for example, that expression e reads the same field of the same object that statement s writes, and that s is executed whenever e is executed (i.e., e dominates s in the control flow graph), then e does not need to be instrumented.

Program	Base Time	Intcpt Ovhd	Unopt Ovhd	Foc Ovhd	Spdup	Frac Ovhd
elevator	0.2	0.14	0.32	0.30	8.6%	46.5%
tsp(12)	0.3	9.59	8.89	8.49	4.5%	46.2%
tsp(14)	0.48	17.06	387.0	393.0	-1.5%	95.7%
hedc	0.6	0.22	0.64	0.55	15.0%	40.0%
moldyn	44.03	1430	172.7	129.5	25.0%	8.1%
montecarlo	15.85	443.2	10.3	0	100%	0%
raytracer	14.34	594	44.8	24.2	46.0%	3.8%

Table 2: Comparison of running time between the optimized algorithm and the block-based algorithm.

The focused reduction-based algorithm [18] completely ignores events in methods shown by the typechecker to be atomic. This greatly reduces the interception overhead as well as the cost of the reduction-based algorithm itself, reducing the median overall slowdown from 38.1 to 1.5 in the experiments in [18]. The opportunity for this more drastic improvement is related to the fact that the reduction-based algorithm is less accurate (*i.e.*, more conservative, producing more false alarms) than the block-based algorithm.

10. RELATED WORK

Choi et al. [7] present a run-time race detection algorithm that uses static analysis to show that some statements cannot be involved in data races and hence do not need to be instrumented. This is similar in spirit to our focused checking. However, there are many situations where type discovery succeeds in showing that a statement is race-free while their static analysis does not. For example, their analysis does not correlate field accesses with thread and synchronization information, so it is unable to show race-freedom for a statement that is executed by multiple threads and accesses a different unshared object in each thread. Also, their analysis does not recognize readonly objects or objects to which there is a unique reference and hence does not recognize that accesses to those objects are race-free. Our technique can verify absence of races in these cases. On the other hand, their static analysis of the weaker-than relation identifies statements that need not be instrumented because another access (that is instrumented) would produce the same warning. This optimization is complementary to and compatible with our approach.

O' Callahan and Choi [17] present a run-time race detection technique that combines lockset-based detection and happens-before-based detection. In their two-phase mode they first run an unoptimized Eraser algorithm and then run a more accurate and more expensive algorithm that monitors only fields on which Eraser reported a potential race. Our work can be combined with theirs by using our focused Eraser algorithm in the first phase. This could also reduce the cost of the second phase, by allowing it to monitor fewer fields, because our focused Eraser algorithm sometimes reports fewer potential races (specifically, fewer false alarms) than full Eraser, as described in Section 6.1.2.

von Praun and Gross [20] reduce the overhead of run-time race detection by using static escape analysis to eliminate some run-time checks, by detecting races at object granularity (instead of field granularity), by conservatively treating method calls as writes to the `this` object and thereby eliminating the need to instrument accesses to fields of `this` (of course, this is a heuristic that can produce more false alarms), and by a variety of lower-level optimizations, described briefly below. While there is some overlap between the benefits of our focused checking and their use of escape analysis, our techniques and theirs are mostly complementary and can be used together. Also, they introduce a modification to the Eraser algorithm designed to reduce the number of false alarms on objects referenced by unique pointers passed between threads; however, their modification can cause some races to be missed (*i.e.*, not reported), while the support for the owner unique in our approach achieves similar benefits in a safe way.

We spent relatively little effort on unrelated lower-level optimizations to our implementation of run-time race detection. For example, in [20], the commonly executed part of the instrumentation is inlined (and the x86 code generator of their ahead-of-time Java compiler is enhanced to handle the inlined code more efficiently than a JVM could [20]), avoiding a method invocation in most cases; we did not do this, because it is harder to implement, but this optimization is compatible with our techniques and could reduce the overhead significantly. In [7], the runtime checking algorithms are implemented in C++ as part of the JVM; this can also give a noticeable constant factor improvement in the overhead compared to our implementation in Java.

Quantitative comparison with these other static-analysis-based techniques for optimizing runtime race detection is very difficult, due to unavailability of the existing implementations of these techniques, differences in the programs and test inputs used as benchmarks (in some cases, we attempted unsuccessfully to obtain the programs used as benchmarks and to find out what test inputs were used), different and non-standard run-time systems (especially for techniques integrated with the compiler or JVM), different lower-level optimizations unrelated to the use of static analysis, etc. Therefore, our comparison with related work is mostly qualitative. As a small first step towards quantitative comparison, the fractional overheads for our combined approach range from 1% to 91%, while the fractional overheads reported in [20] and [7] (for a different set of benchmarks) range from 16% to 129% and 13% to 42% respectively.

Purely static analysis of data races faces a difficult tradeoff between scalability, accuracy (few false alarms), and soundness. For example, RacerX [8] achieves high scalability and accuracy but, unlike race-free type systems, is not sound, i.e., is not guaranteed to detect all races. Race checking by context inference [14] is sound, and it is accurate for a class of programs, but it is less scalable than runtime checking and type discovery, and it does not accurately analyze programs that use heap allocation or arrays. Thus, runtime checking, and combinations of static analysis and runtime checking, are also valuable.

We are not aware of any work on use of static analysis to optimize runtime atomicity checking, other than our recent work on the focused reduction-based algorithm [18].

Acknowledgment. We thank Chandra Boyapati, Cormac Flanagan and Stephen Freund for helpful comments about race-free type systems.

11. REFERENCES

[1] R. Agarwal, A. Sasturkar, and S. D. Stoller. Type discovery for parameterized race-free Java. Technical

Report DAR-04-16, Computer Science Department, SUNY at Stony Brook, Sept. 2004. Available at http://www.cs.sunysb.edu/~stoller/type-discovery/.

[2] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. Technical Report DAR-05-22, Computer Science Department, SUNY at Stony Brook, June 2005. Available at http://www.cs.sunysb.edu/~ragarwal/.

[3] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 149–160. Springer-Verlag, Jan. 2004.

[4] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Laboratory for Computer Science, MIT, Feb. 2004. Available at http://www.eecs.umich.edu/~bchandra/.

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proc. 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 211–230, Nov. 2002.

[6] C. Boyapati and M. C. Rinard. A parameterized type system for race-free Java programs. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 56–69. ACM Press, 2001.

[7] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 258–269. ACM Press, 2002.

[8] D. R. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. pages 237–252. ACM Press, Oct. 2003. Available at http://www.stanford.edu/~engler/.

[9] C. Flanagan and S. Freund. Type-based race detection for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232. ACM Press, 2000.

[10] C. Flanagan and S. Freund. Type inference against races. In *Proc. 11th International Static Analysis Symposium (SAS)*, volume 3148 of *Lecture Notes in Computer Science*. Springer-Verlag, Aug. 2004.

[11] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. 31st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Jan. 2004.

[12] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 338–349. ACM Press, 2003.

[13] K. Havelund. Using runtime analysis to guide model checking of java programs. In *Proc. 7th Int'l. SPIN Workshop on Model Checking of Software*, volume 1885 of *Lecture Notes in Computer Science*, pages 245–264. Springer-Verlag, Aug. 2000.

[14] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Proc. 26th ACM Conference on Programming Language Design and Implementation*. ACM Press, 2004.

[15] Jigsaw — W3C's Server. Available at http://www.w3.org/Jigsaw/.

[16] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[17] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proc. ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2003.

[18] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *Proc. ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, June 2005.

[19] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

[20] C. von Praun and T. R. Gross. Object race detection. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 70–82. ACM Press, Oct. 2001.

[21] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Proc. Third Workshop on Runtime Verification (RV)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2003. A revised and expanded version appeared as Technical Report DAR 04-14, Computer Science Department, SUNY Stony Brook, July 2004 (revised May 2005). Available at http://www.cs.sunysb.edu/~stoller/.