Cooperative Types for Controlling Thread Interference in Java

Jaeheon Yi Tim Disney
UC Santa Cruz, USA

Stephen N. Freund
Williams College, USA

Cormac Flanagan
UC Santa Cruz, USA

ABSTRACT

Multithreaded programs are notoriously prone to unintended interference between concurrent threads. To address this problem, we argue that yield annotations in the source code should document all thread interference, and we present a type system for verifying the absence of undocumented interference in Java programs. Under this type system, well-typed programs behave as if context switches occur only at yield annotations. Thus, well-typed programs can be understood using intuitive sequential reasoning, except where yield annotations remind the programmer to account for thread interference.

Experimental results show that yield annotations describe thread interference more precisely than prior techniques based on method-level atomicity specifications. In particular, yield annotations reduce the number of interference points one must reason about by an order of magnitude. The type system is also more precise than prior methods targeting race freedom, and yield annotations highlight all known concurrency defects in our benchmarks.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification–*formal methods, reliability*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs–*type structure*

General Terms

Languages, Verification, Reliability

Keywords

Cooperability, concurrency, type systems

1. CONTROLLING INTERFERENCE

Developing reliable multithreaded software is challenging due to the potential for nondeterministic interference between concurrent threads. Programmers use synchronization idioms such as locks to control thread interference, but unfortunately do not document where interference may still occur in the source code. Consequently, every programming task (such as a feature extension, code review, etc.) may require the programmer to manually reconstruct the actual interference points by analyzing the synchronization idioms in the source code.

This approach is rather problematic, since manual reconstruction of interference points is tedious and error-prone. Moreover, interference points are fairly sparse in practice, and consequently programmers often simply assume that code is free of interference — a shortcut that is sometimes correct but that may also result in scheduler-dependent defects, which are notoriously difficult to detect or eliminate.

We propose to use yield annotations to document the actual interference points in the source code, thus avoiding the need to manually infer interference points during program maintenance. Moreover, yield annotations enable us to decompose the hard problem of multithreaded program correctness under preemptive scheduling into two simpler correctness requirements:

> *Cooperative-preemptive equivalence*: Does the program exhibit the same behavior under a *cooperative scheduler* (that performs context switches only at yield annotations) as it would under a traditional *preemptive scheduler* (that performs context switches at arbitrary program points)?
>
> *Cooperative correctness*: Is the program correct when run under a cooperative scheduler?

In this paper, we present a static type and effect system for Java that verifies the correctness property of cooperative-preemptive equivalence, which indicates that all thread interference is documented with yield annotations. The type system checks that the instructions of each thread consist of a sequence of *transactions* separated by yield annotations, where each transaction is serializable according to Lipton's theory of reduction [25]. Consequently, any preemptive execution of a well-typed program is guaranteed to behave *as if* executing under a cooperative scheduler, where context switches happen only at explicit yield annotations.

Cooperative scheduling provides an appealing concurrency semantics with the following properties:

- Sequential reasoning is correct by default for code fragments with no yield annotations.
- Thread interference is always highlighted with yields, which remind the programmer to allow for the effects of interleaved concurrent threads.

ISSTA '12, July 15-20, 2012, Minneapolis, MN, USA

```
1  class TSP {
2    final Object lock;
3    volatile int shortestPathLength;
4
5    compound void searchFrom(Path path) {
6      if (path.length >= this..shortestPathLength)
7        return;
8
9      if (path.isComplete()) {
10       ..synchronized (this.lock) {
11         if (path.length < this.shortestPathLength)
12           this.shortestPathLength = path.length;
13       }
14     } else {
15       for (Path c: path.children())
16         this..searchFrom#(c);
17     }
18   }
19 }
```

Figure 1: Traveling Salesperson Algorithm

A previous user study showed that these properties provide a distinct benefit to the programmer [33]. Specifically, the presence of yield annotations produces a statistically significant improvement in the ability of programmers to identify concurrency defects during code inspection.

We have developed a prototype implementation, called the Java Cooperability Checker (JCC), of our type system for verifying cooperative-preemptive equivalence in Java. Experimental results on a range of Java benchmarks show that JCC requires yield annotations on only about 13 interference points (IPs) per KLOC. In comparison, previous forms of non-interference specifications generate significantly more interference points per KLOC: 139 IP/KLOC using atomic methods specifications; 95 IP/KLOC when reasoning about race conditions; and 32 IP/KLOC when reasoning about both races and atomic methods.

Example. To illustrate the benefits of explicit yield annotations, consider the traveling salesperson algorithm shown in Figure 1. The `TSP` class contains a method `searchFrom` that recursively searches through all extensions of a particular `path`, aborting the search whenever `path.length` becomes greater than `shortestPathLength` (the length of the shortest complete path found so far). To exploit multicore processors, multiple threads may concurrently invoke `searchFrom` on the same `TSP` object. The variable `shortestPathLength` is protected by the mutex `lock` for all writes, but racy reads are permitted to maximize performance. Consequently, accesses to `shortestPathLength` on lines 6 and 12 are racy, but the read on line 11 is race-free, since `lock` is held.

The code uses the notation "`..`" as a lightweight syntax for yield annotations. The "`..`" on line 6 indicates that interference may occur before the read of `shortestPathLength` on line 6 (because of concurrent writes) and before the lock acquire at line 10 (because a concurrent call to `searchFrom` may also be trying to acquire that lock). Note that the potentially racy write to `shortestPathLength` on line 12 does not require a preceding yield, since a reducible transaction may contain one racy or non-mover operation under Lipton's theory [25].

To facilitate modular reasoning, the method `searchFrom` is declared `compound`, meaning that it contains internal yield points. The method call on line 16 is highlighted with a postfix "`#`" to indicate that the callee is `compound`, and so invariants over shared state that hold before the call may not hold after the call returns. That call is also a yielding call (as indicated by the notation "`..`") because interference from other threads may become visible in between consecutive calls to that method at run time.

This work uses the yield annotation syntax "`..`" in preference to the "`yield`" statements of prior work [3, 4, 6, 22, 40, 41] to more precisely characterize *why* interference may occur. For example, the annotation at line 6 of Figure 1 precisely documents that interference occurs on the read of `shortestPathLength`. In comparison, in the following code the `yield` statement suggests that one of the subsequent reads of `path.length` or `shortestPathLength` is interfering or racy, but it is not immediately obvious which one.

```
compound void searchFrom(Path path) {
  yield;
  if (path.length >= shortestPathLength)
  ...
}
```

Contributions. This paper extends a prior type system for checking yield annotations in an imperative variant of the λ-calculus [40]. Adapting those ideas to a realistic object-oriented programming language requires a number of enhancements and extensions, including a more flexible and precise notion of "yield" and an effect system that reasons about, for example, conditional effects.

We also present the first implementation of a type-based cooperability checker. Our experimental results validate that cooperability is an effective way to capture and reason about thread interference, and we show that the use of yield annotations dramatically reduces the number of interference points that must be considered by the programmer.

In summary, the contributions of our work are as follows:

- We present a type system for verifying lightweight, precise non-interference specifications in multithreaded software (Section 4).
- We show that this type system satisfies the standard preservation property, and ensures that well-typed programs are cooperative-preemptive equivalent.
- We describe an implementation for Java (Section 5).
- For our benchmarks, the type system identified just 13 interference points per KLOC, a significant improvement over prior non-interference specifications based on atomic methods (139), race-freedom (95), or a combination of atomic methods and race freedom (32).
- Our experimental results also show that yield annotations highlight all known concurrency bugs in our benchmark suite (Section 6).

2. THE LANGUAGE YIELDJAVA

We formalize our ideas in terms of the idealized language YieldJava, a multithreaded subset of Java with yield annotations. The YieldJava syntax is summarized in Figure 2.

$$
\begin{array}{rcl}
P \in \mathit{Program} & = & \overline{\mathit{defn}} \\
\mathit{defn} \in \mathit{Definition} & ::= & \texttt{class}\ c\ \texttt{\{}\ \overline{\mathit{field}}\ \ \overline{\mathit{meth}}\ \texttt{\}} \\
\mathit{field} \in \mathit{Field} & ::= & c\ f \\
\mathit{meth} \in \mathit{Method} & ::= & a\ c\ m\texttt{(}\overline{c\ x}\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \\
f \in \mathit{FieldName} & = & \mathit{Normal} \cup \mathit{Final} \cup \mathit{Volatile} \\
e, \ell \in \mathit{Expr} & ::= & x \mid \texttt{null} \\
 & & \mid e_\gamma f \mid e_\gamma f = e \\
 & & \mid e_\gamma m\texttt{(}\overline{e}\texttt{)} \mid e_\gamma m\texttt{\#(}\overline{e}\texttt{)} \\
 & & \mid \texttt{new}\ c\texttt{(}\overline{e}\texttt{)} \mid e_\gamma\texttt{sync}\ e \\
 & & \mid \texttt{fork}\ e \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e \\
 & & \mid \texttt{if}\ e\ e\ e \mid \texttt{while}\ e\ e \\
\\
\gamma \in \mathit{OptYield} & ::= & \texttt{.} \mid \texttt{..} \\
c, d \in \mathit{ClassName} & & \\
x, y \in \mathit{Var} & & \\
m \in \mathit{MethodName} & & \\
a \in \mathit{Effect} & &
\end{array}
$$

Figure 2: YIELDJAVA **Syntax**

A program P is a sequence of class definitions. Each definition associates a name c with a collection of field and method declarations. A field declaration includes a type and a name. Field names are syntactically divided into three categories:

Normal fields are mutable and free of race conditions.

Final fields are immutable and thus race-free.

Volatile fields are mutable and may have concurrent conflicting accesses.

We assume that race freedom for *Normal* fields is verified separately by, for example, a race-free type system [7, 18, 1]. (Our implementation does support racy accesses to non-`volatile` fields, as described in Section 5.)

Each method declaration $a\ c\ m\texttt{(}\overline{c\ x}\texttt{)}\ \texttt{\{}\ e\ \texttt{\}}$ defines a method m with return type c that takes parameters $\overline{x}$ of type $\overline{c}$. The method declaration also includes an effect a, whose syntax and semantics are described in Section 3 below.

A field read expression $e_\gamma f$ comes in two forms, $e\texttt{.}f$ and $e\texttt{..}f$, depending on the optional yield annotation γ, and similarly for field writes $e_\gamma f = e$. The yielding read $e\texttt{..}f$ evaluates the subexpression e to an object reference, performs a yield, and then reads the f field of the referenced object. Method calls $e_\gamma m\texttt{(}\overline{e}\texttt{)}$ also include an optional yield, and calls to `compound` or non-atomic methods must be marked as $e_\gamma m\texttt{\#(}\overline{e}\texttt{)}$. We also support a synchronized block $e_1{}_\gamma\texttt{sync}\ e_2$, which is analogous to Java's synchronized statement

$$\texttt{synchronized (}e_1\texttt{) \{ }e_2\texttt{ \}}$$

and again supports an optional yield. The following table summarizes which constructs may be annotated with yields.

Expression Form	Non-Yielding	Yielding
Field Read	$e\texttt{.}f$	$e\texttt{..}f$
Field Write	$e\texttt{.}f = e$	$e\texttt{..}f = e$
Atomic Method Call	$e\texttt{.}m\texttt{(}\overline{e}\texttt{)}$	$e\texttt{..}m\texttt{(}\overline{e}\texttt{)}$
Non-Atomic Method Call	$e\texttt{.}m\texttt{\#(}\overline{e}\texttt{)}$	$e\texttt{..}m\texttt{\#(}\overline{e}\texttt{)}$
Lock Synchronization	$e\texttt{.sync}\ e$	$e\texttt{..sync}\ e$

Finally, an object allocation $\texttt{new}\ c\texttt{(}\overline{e}\texttt{)}$ creates a new object of type c and initializes its fields to the values of the expression sequence $\overline{e}$. We assume that all programs include an empty class `Unit`. The language includes the special constant `null`, which has any class type, including `Unit`.

3. EFFECTS FOR COOPERABILITY

Our effect system characterizes the behavior of each program subexpression using two kinds of effects: *mover effects* and *atomicity effects*.

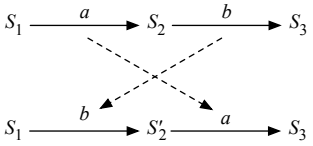
Mover Effects. A *mover effect* μ characterizes the behavior of a program expression in terms of how operations of that expression commute with operations of other threads:

$$\mu \quad ::= \quad \text{F} \mid \text{Y} \mid \text{M} \mid \text{R} \mid \text{L} \mid \text{N}$$

F: The effect F (for functional) describes expressions whose result does not depend on any mutable state. Hence, re-evaluating a functional expression is guaranteed to produce the same result. (Our type system requires all lock names to be functional to ensure that it does not confuse distinct locks.)

Y: The effect Y describes yield operations, denoted as "`..`". These expressions mark transactional boundaries where the current transaction ends and a new one starts.

R: The effect R describes *right-mover* expressions such as lock acquires. In more detail, suppose a trace contains an acquire operation a that is immediately followed by an operation b of a different thread. Then the two operations a and b *commute* and can be swapped without changing the overall behavior or final state of the trace, as illustrated by the commuting diagram below. Thus, we consider acquire operations to be *right-movers*.



L: The effect L describes left-mover expressions such as lock releases that commute to the left across a preceding operation of a different thread (such as operation b in the above diagram).

M: The effect M describes both-mover expressions such as race-free variable accesses that commute both left and right with operations by other threads.

N: The effect N describes non-mover code that may perform a racy access or contain right-movers followed by left-movers (as in a synchronized block).

Suppose the sequence of instructions α performed by a thread consists of: (1) zero or more right-movers, followed by (2) at most one non-mover, followed by (3) zero or more left-movers. Then, according to Lipton's theory of reduction [25], instructions of other threads that are interleaved into α can be commuted out so that α executes serially, without interleaved operations of other threads. In this case, we consider α a *reducible transaction*, or simply a *transaction*.

Our type system ensures that the instructions performed by each thread consist of reducible transactions separated by yield annotations. The type system works by composing the effects for individual operations according to the following iterative closure (μ^*) and sequential composition ($\mu_1; \mu_2$) operations. These operations are partial and may fail if the code between two successive yields is not reducible (indicated with a "$-$"). For example, the sequential composition

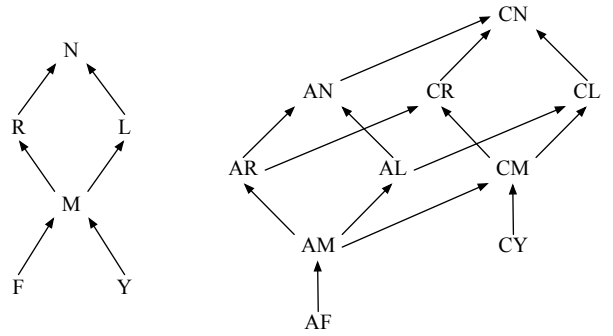


Figure 3: Mover and Combined Effect Lattices

(L; R) is undefined, indicating code containing a left-mover followed by a right-mover does not form a transaction.

μ	μ^*
F	F
Y	M
M	M
R	R
L	L
N	—

;	F	Y	M	R	L	N
F	F	Y	M	R	L	N
Y	Y	Y	Y	Y	L	L
M	M	Y	M	R	L	N
R	R	R	R	R	N	N
L	L	Y	L	—	L	—
N	N	R	N	—	N	—

Composition with Y is defined to capture how yielding may be used. For example, "Y; N" is L, to ensure that "N; Y; N" is permissible. Mover effects are ordered by the relation $\sqsubset$ captured by the first lattice in Figure 3.

Atomicity Effects. Each program expression also has an *atomicity effect* $\tau \in \{\text{A}, \text{C}\}$ that is either: A (for atomic) if the expression never yields, or C (for compound) if the expression may yield. Ordering ($\sqsubset$), iterative closure (τ^*) and sequential composition ($\tau_1; \tau_2$) for atomicity effects are defined by:

$$\text{A} \sqsubset \text{C} \qquad \tau^* \stackrel{\text{def}}{=} \tau \qquad \tau_1; \tau_2 \stackrel{\text{def}}{=} \tau_1 \sqcup \tau_2$$

Combined Effects. A *combined effect* κ is a pair $\tau\mu$ of an atomicity effect τ and a mover effect μ. Note that not all combined effects are meaningful. In particular, AY and CF are contradictory: an atomic piece of code may not contain a yield, and code with yields cannot be considered functional. We define the ordering relation and the join, iterative closure, and sequential composition operations on combined effects in a point-wise manner, and the diagram in Figure 3 summarizes the resulting lattice of combined effects.

$$\begin{aligned}
\kappa &::= \tau\,\mu \\
\tau_1\,\mu_1 \sqsubseteq \tau_2\,\mu_2 &\quad \text{iff} \quad \tau_1 \sqsubseteq \tau_2 \text{ and } \mu_1 \sqsubseteq \mu_2 \\
\tau_1\,\mu_1 \sqcup \tau_2\,\mu_2 &\stackrel{\text{def}}{=} \tau_3\,\mu_3 \quad \text{where } \tau_3 = \tau_1 \sqcup \tau_2,\ \mu_3 = \mu_1 \sqcup \mu_2 \\
\tau_1\,\mu_1\,;\,\tau_2\,\mu_2 &\stackrel{\text{def}}{=} \tau_3\,\mu_3 \quad \text{where } \tau_3 = \tau_1; \tau_2,\ \mu_3 = \mu_1; \mu_2 \\
(\tau\mu)^* &\stackrel{\text{def}}{=} \tau^*\mu^*
\end{aligned}$$

Conditional Effects. As outlined above, the effect of acquiring a lock `m` is AR, since an acquire is a right-mover containing no yields. If the lock `m` is already held by the current thread, however, then the re-entrant lock acquire is actually a no-op and is more precisely characterized as an atomic both-mover AM.

We introduce *conditional effects* to capture situations like this where the effect of an operation depends on which locks are held by the current thread. We use ℓ to range over expressions that are functional (F) and which always denote the same lock. An *effect* a is then either a combined effect κ or a conditional effect $\ell\,?\,a_1 : a_2$, which is equivalent to a_1 if the lock ℓ is held, and is equivalent to a_2 otherwise. We extend the iterative closure, sequential composition, and join operations to conditional effects as follows:

$$\begin{aligned}
a &::= \kappa \mid \ell\,?\,a_1 : a_2 \\
(\ell\,?\,a_1 : a_2) \sqcup a &= \ell\,?\,(a_1 \sqcup a) : (a_2 \sqcup a) \\
a \sqcup (\ell\,?\,a_1 : a_2) &= \ell\,?\,(a \sqcup a_1) : (a \sqcup a_2) \\
(\ell\,?\,a_1 : a_2)^* &= \ell\,?\,a_1^* : a_2^* \\
(\ell\,?\,a_1 : a_2); a &= \ell\,?\,(a_1; a) : (a_2; a) \\
a; (\ell\,?\,a_1 : a_2) &= \ell\,?\,(a; a_1) : (a; a_2)
\end{aligned}$$

We extend the effect ordering to conditional effects in a manner similar to earlier type systems for atomicity [15]. To decide $a_1 \sqsubseteq a_2$, we use an auxiliary relation $\sqsubseteq_n^h$, where h is a set of locks known to be *h*eld by the current thread, and n is a set of locks known *n*ot to be held by the current thread. We define $a_1 \sqsubseteq a_2$ to be $a_1 \sqsubseteq_\emptyset^\emptyset a_2$ and check $a_1 \sqsubseteq_n^h a_2$ recursively as follows:

$$\frac{\kappa_1 \sqsubseteq \kappa_2}{\kappa_1 \sqsubseteq_n^h \kappa_2} \qquad \frac{\ell \notin n \Rightarrow a_1 \sqsubseteq_n^{h \cup \{l\}} a \qquad \ell \notin h \Rightarrow a_2 \sqsubseteq_{n \cup \{l\}}^h a}{\ell\,?\,a_1 : a_2 \sqsubseteq_n^h a} \qquad \frac{\ell \notin n \Rightarrow \kappa \sqsubseteq_n^{h \cup \{l\}} a_1 \qquad \ell \notin h \Rightarrow \kappa \sqsubseteq_{n \cup \{l\}}^h a_2}{\kappa \sqsubseteq_n^h \ell\,?\,a_1 : a_2}$$

4. TYPE AND EFFECT SYSTEM

The YieldJava type system ensures that each thread consists of reducible transactions separated by yields, and consequently that thread interference is observable only at yield annotations. The core of the type system is a set of rules for reasoning about the effect of an expression, as captured by the judgment:

$$P; E \vdash e : c \cdot a$$

where e is an expression of type c, and a is an effect describing the behavior of e. The program P is included to provide access to class declarations, and the environment E maps free variables in e to their types. Figure 4 presents the complete set of rules for expressions, as well as auxiliary rules to reason about methods, classes, effects, and so on. We describe the most important rules defining these judgments:

[EXP VAR] All variables are immutable in YieldJava and thus have cooperability effect AF, since a variable access is atomic and evaluates to a constant value.

[EXP NEW] The object allocation rule first retrieves the definition of the class c from P and ensures the arguments $e_{1..n}$ match the field types from c. The effect of the whole expression is the composition of effects of evaluating $e_{1..n}$ composed with the effect AM, reflecting that `new` is not functional (since re-evaluating an object allocation would not return the same object).

[EXP REF] This rule handles a read $e.f$ of a *Normal* or *Final* field. If f is *Normal* and thus race-free, the field access has effect AM as it commutes with steps by other threads. If f is *Final*, the access has the more precise effect AF.

[EXP REF RACE] A racy read $e_\gamma f$ of a volatile field may be annotated with a yield point (if $\gamma =$ "`.`") or not (if $\gamma =$ "`·`").

$\boxed{P; E \vdash e : c \cdot a}$

[EXP VAR]
$$\frac{P \vdash E \quad E = E_1, c\ x, E_2}{P; E \vdash x : c \cdot \text{AF}}$$

[EXP NULL]
$$\frac{P \vdash E \quad \texttt{class}\ c\ \{\ \ldots\ \} \in P}{P; E \vdash \texttt{null} : c \cdot \text{AF}}$$

[EXP NEW]
$$\frac{\texttt{class}\ c\ \{\ d_i\ x_i\ ^{i \in 1..n}\ \ldots\ \} \in P \quad P; E \vdash e_i : d_i \cdot a_i \quad \forall i \in 1..n}{P; E \vdash \texttt{new}\ c(e_{1..n}) : c \cdot (a_1; \cdots; a_n; \text{AM})}$$

[EXP REF]
$$\frac{P; E \vdash e : c \cdot a \quad \texttt{class}\ c\ \{\ldots d\ f \ldots\} \in P \quad (f \in \mathit{Normal} \wedge a' = \text{AM}) \vee (f \in \mathit{Final} \wedge a' = \text{AF})}{P; E \vdash e.f : d \cdot (a; a')}$$

[EXP ASSIGN]
$$\frac{P; E \vdash e : c \cdot a \quad P; E \vdash e' : d \cdot a' \quad \texttt{class}\ c\ \{\ldots d\ f \ldots\} \in P \quad f \in \mathit{Normal}}{P; E \vdash (e.f = e') : d \cdot (a; a'; \text{AM})}$$

[EXP REF RACE]
$$\frac{P; E \vdash e : c \cdot a \quad \texttt{class}\ c\ \{\ldots d\ f \ldots\} \in P \quad f \in \mathit{Volatile}}{P; E \vdash e_\gamma f : d \cdot (a; [\![\gamma]\!]; \text{AN})}$$

[EXP ASSIGN RACE]
$$\frac{P; E \vdash e : c \cdot a \quad P; E \vdash e' : d \cdot a' \quad \texttt{class}\ c\ \{\ \ldots\ d\ f\ \ldots\ \} \in P \quad f \in \mathit{Volatile}}{P; E \vdash (e_\gamma f = e') : d \cdot (a; a'; [\![\gamma]\!]; \text{AN})}$$

[EXP CALL ATOMIC]
$$\frac{\begin{array}{c} P; E \vdash e : c \cdot a \quad \texttt{class}\ c\ \{\ldots \mathit{meth} \ldots\} \in P \\ \mathit{meth} = a'\ c'\ m(d_i\ x_i\ ^{i \in 1..n})\ \{\ e'\ \} \\ P; E \vdash a'[\texttt{this} := e, x_i := e_i\ ^{i \in 1..n}] \uparrow a'' \quad P; E \vdash e_i : d_i \cdot a_i \quad \forall i \in 1..n \quad a'' \sqsubseteq \text{AN} \end{array}}{P; E \vdash e_\gamma m(e_{1..n}) : c' \cdot (a; a_1; \cdots; a_n; [\![\gamma]\!]; a'')}$$

[EXP CALL COMPOUND]
$$\frac{\begin{array}{c} P; E \vdash e : c \cdot a \quad \texttt{class}\ c\ \{\ldots \mathit{meth} \ldots\} \in P \\ \mathit{meth} = a'\ c'\ m(d_i\ x_i\ ^{i \in 1..n})\ \{\ e'\ \} \\ P; E \vdash a'[\texttt{this} := e, x_i := e_i\ ^{i \in 1..n}] \uparrow a'' \quad P; E \vdash e_i : d_i \cdot a_i \quad \forall i \in 1..n \end{array}}{P; E \vdash e_\gamma m\texttt{\#}(e_{1..n}) : c' \cdot (a; a_1; \cdots; a_n; [\![\gamma]\!]; a'')}$$

[EXP SYNC]
$$\frac{P; E \vdash_{\text{lock}} \ell \quad P; E \vdash e : c \cdot a}{P; E \vdash \ell_\gamma \texttt{sync}\ e\ : c \cdot \mathcal{S}(\ell, \gamma, a)}$$

[EXP FORK]
$$\frac{P; E \vdash e : c \cdot a}{P; E \vdash \texttt{fork}\ e : \texttt{Unit} \cdot \text{AL}}$$

[EXP WHILE]
$$\frac{P; E \vdash e_1 : c_1 \cdot a_1 \quad P; E \vdash e_2 : c_2 \cdot a_2}{P; E \vdash \texttt{while}\ e_1\ e_2 : \texttt{Unit} \cdot (a_1; (a_2; a_1)^*)}$$

[EXP IF]
$$\frac{\begin{array}{c} P; E \vdash e_1 : d \cdot a_1 \quad P; E \vdash e_i : c \cdot a_i \quad \forall i \in 2..3 \\ a' = a_1; (a_2 \sqcup a_3) \end{array}}{P; E \vdash \texttt{if}\ e_1\ e_2\ e_3 : c \cdot a'}$$

[EXP LET]
$$\frac{P; E \vdash e_1 : c_1 \cdot a_1 \quad P; E, c_1\ x \vdash e_2 : c_2 \cdot a_2 \quad P; E \vdash a_2[x := e_1] \uparrow a_2'}{P; E \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : c_2 \cdot (a_1; a_2')}$$

$\boxed{P \vdash E}$

[ENV ϵ]
$$\frac{}{P \vdash \epsilon}$$

[ENV X]
$$\frac{P \vdash E \quad x \notin \mathit{dom}(E) \quad \texttt{class}\ c\ \{\ \ldots\} \in P}{P \vdash (E, c\ x)}$$

$\boxed{P; E \vdash a}$

[AT BASE]
$$\frac{P \vdash E}{P; E \vdash \kappa}$$

[AT COND]
$$\frac{P; E \vdash_{\text{lock}} \ell \quad P; E \vdash a_i \quad \forall i \in 1..2}{P; E \vdash \ell\,?\,a_1 : a_2}$$

$\boxed{P; E \vdash_{\text{lock}} e}$

[LOCK EXP]
$$\frac{P; E \vdash e : c \cdot \text{AF}}{P; E \vdash_{\text{lock}} e}$$

$\boxed{P; E \vdash a \uparrow a'}$

[LIFT BASE]
$$\frac{P \vdash E}{P; E \vdash \kappa \uparrow \kappa}$$

[LIFT LOCK]
$$\frac{\begin{array}{c} P; E \vdash a_i \uparrow a_i' \quad \forall i \in 1..2 \\ P; E \vdash_{\text{lock}} \ell \implies a'' = \ell\,?\,a_1' : a_2' \\ P; E \not\vdash_{\text{lock}} \ell \implies a'' = a_1' \sqcup a_2' \end{array}}{P; E \vdash (\ell\,?\,a_1 : a_2) \uparrow a''}$$

$\boxed{P; E \vdash \mathit{meth}}$

[METHOD]
$$\frac{\begin{array}{c} P; E, \overline{d\ x} \vdash e : c \cdot a' \quad P; E, \overline{d\ x} \vdash a \quad a' \sqsubseteq a \end{array}}{P; E \vdash a\ c\ m(\overline{d\ x})\ \{\ e\ \}}$$

$\boxed{P \vdash \mathit{defn}}$

[CLASS]
$$\frac{\begin{array}{c} \mathit{field}_i = d_i\ f_i \quad \forall i \in 1..m \quad \texttt{class}\ d_i\ \{\ \ldots\ \} \in P \quad \forall i \in 1..m \\ P; c\ \texttt{this} \vdash \mathit{meth}_i \quad \forall i \in 1..n \end{array}}{P \vdash \texttt{class}\ c\ \{\ \mathit{field}_{1..m}\ \mathit{meth}_{1..n}\ \}}$$

$\boxed{P \vdash \text{OK}}$

[PROGRAM]
$$\frac{P = \mathit{defn}_{1..n} \quad P \vdash \mathit{defn}_i \quad \forall i \in 1..n \quad \mathit{ClassesOnce}(P) \quad \mathit{FieldsOnce}(P) \quad \mathit{MethodsOnce}(P)}{P \vdash \text{OK}}$$

Figure 4: YieldJava **Type Rules**

We use the auxiliary function $[\![\gamma]\!]$ to map γ to the corresponding effect, where $[\![.]\!] = \text{AF}$ and $[\![..]\!] = \text{CY}$. Thus, if the expression e has effect a, then the non-yielding racy access $e.f$ has effect $(a; \text{AF}; \text{AN})$, whereas the yielding racy access $e..f$ has effect $(a; \text{CY}; \text{AN})$.

[EXP SYNC] The rule for the synchronized statement ℓ_γ`sync` e first checks that ℓ is a valid lock expression ($P; E \vdash_{\text{lock}} \ell$), meaning that ℓ must have effect AF to guarantee that it always denotes the same lock at run time.

The rule then computes the effect $\mathcal{S}(\ell, \gamma, a)$, where a is the effect of e, and γ specifies whether there is a yield point. The function S is defined as:

a	$\mathcal{S}(\ell, \gamma, a)$
κ	$\ell \,?\, \kappa : ([\![\gamma]\!]; \text{AR}; \kappa; \text{AL})$
$\ell \,?\, a_1 : a_2$	$\mathcal{S}(\ell, \gamma, a_1)$
$\ell' \,?\, a_1 : a_2$	$\ell' \,?\, \mathcal{S}(\ell, \gamma, a_1) : \mathcal{S}(\ell, \gamma, a_2)$ if $\ell \neq \ell'$

If the synchronized body e has a basic effect κ and the lock ℓ is already held, then the synchronized statement also has effect κ, since the acquire and release operations are no-ops. Note that in this case the yield operation is ignored, since it is unnecessary.

If e has effect κ and the lock is not already held, then the synchronized statement has effect $([\![\gamma]\!]; \text{AR}; \kappa; \text{AL})$, since the execution consists of a yield point, followed by a right-mover (the acquire), followed by κ (the body), followed by a left-mover (the release).

If e has conditional effect $\ell \,?\, a_1 : a_2$, where ℓ is the lock being acquired by this synchronized statement, then we ignore a_2 and recursively apply S to a_1, since ℓ is held within e.

Finally, if e has an effect that is conditional on some other lock ℓ', then we recursively apply S to both branches.

[EXP LET] This rule for `let` x `=` e_1 `in` e_2 infers effects a_1 and a_2 for e_1 and e_2, respectively. Care must be taken when constructing the effect for the let-expression because a_2 may refer to the let-bound variable x. For example, the body of the following `let` expression produces an effect that is conditional on whether the lock `x` is held.

```
let x = e1 in x.sync ...
```

Thus, we apply the substitution $[x := e_1]$ to yield a corresponding effect $a_2[x := e_1]$ that does not mention x. However, e_1 may not have effect AF, in which case $a_2[x := e_1]$ may not be valid (because it could contain e_1 as part of a non-constant lock expression). As in our previous work [15], we use the judgment

$$P; E \vdash a_2[x := e_1] \uparrow a_2'$$

to *lift* the effect $a_2[x := e_1]$ to a well-formed effect a_2' that is greater than or equal to $a_2[x := e_1]$.

[EXP CALL ATOMIC] This rule handles calls to atomic methods. The callee's declared effect a' may refer to `this` or parameters $x_{1..n}$. Therefore, we substitute

- the actual receiver e for `this` and
- the actual arguments $e_{1..n}$ for the parameters $x_{1..n}$

to produce the effect $a'[\texttt{this} := e, x_i := e_i \ ^{i \in 1..n}]$, and ensure that the resulting effect is valid by lifting it to an effect a'' that is well-formed in the current environment. Effect a'' must be an atomic effect and less than or equal to AN.

The rule for a compound method invocation $e_\gamma m$`#(`$e_{1..n}$`)` is similar, but removes the requirement that the computed effect a'' is atomic.

[EXP FORK] A fork expression `fork` e creates a new thread to evaluate e. Since a fork operation cannot commute past the operations of its child thread, fork operations are left movers.

[METHOD], [CLASS], and [PROG] These rules verify the basic well-formedness requirements of methods, classes, and programs. The [PROG] rule uses additional predicates to ensure no class is declared twice (*ClassesOnce*(P)), no field name is declared twice in a class (*FieldsOnce*(P)), and no method name is declared twice in a class (*MethodsOnce*(P)). These are defined more precisely in [17].

Correctness. The extended version of this paper presents a formal semantics for YieldJava [39]. A run-time state σ extends the program's source code with dynamically allocated objects and dynamically created threads. We present two variants of our operational semantics: a preemptive semantics ($\rightarrow$) that context switches at arbitrary program points, and a cooperative semantics ($\rightarrow_c$) that context switches only at yield annotations. As expected, the cooperative semantics is more restrictive than the preemptive, thus and ($\rightarrow_c$) $\subset$ ($\rightarrow$). Both satisfy the standard preservation property that evaluation preserves well-typing (under an appropriate extension of the type system to run-time states, denoted $\vdash \sigma$).

THEOREM 1 (PRESERVATION) *If* $\vdash \sigma$ *and* $\sigma \rightarrow \sigma'$ *then* $\vdash \sigma'$.

The more interesting correctness theorem is that a well-typed program exhibits equivalent behavior under both semantics. We say a state σ is *yielding* if no thread in σ is in the middle of a transaction. If a well-typed yielding state σ can reach a yielding state σ' under the preemptive semantics, it can also reach that state under the cooperative semantics.

THEOREM 2 (COOPERATIVE-PREEMPTIVE EQUIVALENCE) *If* $\vdash \sigma$ *and* $\sigma \rightarrow^* \sigma'$ *then* $\sigma \rightarrow_c^* \sigma'$, *provided* σ *and* σ' *are yielding.*

Therefore, we can reason about the correctness of well-typed programs under the simpler cooperative semantics ($\rightarrow_c$), since this correctness result also applies to executions under the preemptive semantics ($\rightarrow$). Note that the converse to this theorem also holds, since ($\rightarrow_c$) $\subset$ ($\rightarrow$).

5. IMPLEMENTATION

We have developed an implementation called JCC that extends the YieldJava type system to support the Java language.[1] JCC uses the standard Java modifiers `final` and

[1]Our implementation does not currently support generic classes due to limitations in the front-end checker upon which JCC is built, but supporting generic types is not fundamentally problematic.

`volatile` to classify fields as either *Final* or *Volatile*; all other fields are considered *Normal*. We introduce one new modifier, `racy`, to capture intentionally racy *Normal* fields. JCC assumes that correct field annotations are provided for the input program. Such annotations could be generated using RCC/JAVA [1] or any other analysis technique. For our experiments, we leveraged that tool, as well as the FASTTRACK [14] race detector, to identify racy fields.

JCC supports annotations on methods to describe their effects. The following three effect keywords are sufficient to annotate most methods:

`atomic:` an atomic non-mover method with effect AN.

`mover:` an atomic both-mover method with effect AM.

`compound:` a compound non-mover method with effect CN.

We support annotations to describe all elements of the conditional effect lattice, such as `(atomic left-mover)`, as well. These effect annotations may be combined to form conditional effects, as in `(this ? mover : compound)`. Method effect annotations appear alongside standard method modifiers, as in the following illustrative examples:

```
atomic void m1() { ... }

atomic left-mover void m2() { ... }

lock ? mover : atomic void m3() { ... }

lock ? (atomic left-mover) : (compound non-mover)
void m4() { ... }
```

As in YIELDJAVA, field accesses and method invocations may be written using "`..`" in place of "`.`" to indicate interference points. Yielding synchronized statements use the syntax "`..synchronized(e) { ... }`".

Given a program with cooperability annotations, JCC reports a warning whenever interference may occur at a program point not corresponding to a yield, and whenever a method's specification is not satisfied by its implementation.

Type System Extensions. JCC extends YIELDJAVA to support subtyping, subclassing, and method overriding. It allows the effect of an overriding method to change covariantly, and so requires that $b \sqsubseteq a$ in the following:

```
class C           {  a t m() { ... }  }
class D extends C {  b t m() { ... }  }
```

To reduce the burden of annotating code with cooperability effects, JCC uses carefully chosen defaults when annotations are absent. In particular, unannotated fields are race free, and unannotated methods are atomic both-movers.

Although data races should in general be avoided, large programs often have some intentional races, which JCC supports via a `racy` annotation on *Normal* fields. A read from a racy field must be written as e`..`f`#`. Here, the double dots as usual indicate a yield point, and the trailing `#` identifies the racy nature of the read (and that the programmer needs to consider the consequences of Java's relaxed memory model [26]). The overall effect of e`..`f`#` is the composition of a yield and a non-mover memory access: $(\text{CY}; \text{AN}) = \text{CL}$. Writes are similar.

The YIELDJAVA checker handles array accesses similar to *Normal* fields. Note that in Java, array elements are never `final` or `volatile`. Racy array accesses must be annotated with "`#`" and are assumed to be yield points.

Table 1: Effects for Other Access Forms.

Access Type	Syntax	Effect
racy read	e_1`..`f`#`	$a_1; \text{CL}$
racy write	e_1`..`f`#` $= e_2$	$a_1; a_2; \text{CL}$
race-free array read	$e_1[e_2]$	$a_1; a_2; \text{AM}$
race-free array write	$e_1[e_2] = e_3$	$a_1; a_2; a_3; \text{AM}$
racy array read	$e_1[e_2]$`#`	$a_1; a_2; \text{CL}$
racy array write	$e_1[e_2]$`#` $= e_3$	$a_1; a_2; a_3; \text{CL}$
write-guarded read with lock held	$e_1.f$	$a_1; \text{AM}$
write-guarded read without lock held	e_1`..`f	$a_1; \text{AN}$
write-guarded write with lock held	$e_1.f = e_2$	$a_1; a_2; \text{AN}$

YIELDJAVA also supports write-guarded fields (such as `shortestPathLength` from Figure 1) for which a protecting lock is held for all writes but not necessarily for reads. In this case, a read while holding the protecting lock is a both-mover, since there can be no concurrent writes. However, a write with the lock held is a non-mover, since there may be concurrent reads that do not hold the lock. We summarize these extra rules in Table 1.

6. EXPERIMENTAL EVALUATION

We applied JCC to benchmark programs including a number of standard library classes from Java 1.4; `sparse`, `raytracer`, `sor`, and `moldyn` from the Java Grande suite [23]; `tsp`, a solver for the traveling salesman problem [36]; and `elevator`, a real-time discrete event simulator [36]. These programs use a variety of synchronization idioms, and previous work has revealed a number of interesting concurrency bugs in these programs. Thus, they show the ability of our annotations to capture thread interference under various conditions and to highlight unintended, problematic interference. Three of these programs (`raytracer`, `sor`, and `moldyn`) use broken barrier implementations [14]. We discuss those problems below and use versions with corrected barrier code (named `raytracer-fixed`, `sor-fixed`, and `moldyn-fixed`) in our experiments. We performed all experiments on a 2 GHz dual-core computer with 3 GB memory, using the Java 1.6.0 HotSpot 64-bit Server VM. The checker analyzed each benchmark in under 2 seconds.

Figure 2 shows the size of each benchmark program, the time required to manually insert JCC annotations, and the number of annotations required to enable successful type checking. This count includes all `racy` field annotations, method specifications, and occurrences of "`..`" and "`#`". Even for programs comprising several thousand lines, the annotation burden is quite low. Each program was annotated and checked in about 10 to 30 minutes, and roughly one annotation per 30 lines of code was required. We did have some previous experience using these programs, which facilitated the annotation process, but since we intend JCC to be used during development, we believe our experience reflects the cost incurred by the intended use of our technique.

Table 2: Interference Points and Unintended Yields

Program	Size (lines)	Annotation Time (min)	Annotation Count	Interference Points					Unintended Yields
				NoSpec	Race	Atomic	AtomicRace	Yields	
java.util.zip.Inflater	317	9	4	36	12	0	0	0	0
java.util.zip.Deflater	381	7	8	49	13	0	0	0	0
java.lang.StringBuffer	1,276	20	10	210	81	9	2	1	1
java.lang.String	2,307	15	5	230	87	6	2	1	0
java.io.PrintWriter	534	40	109	73	99	130	97	26	9
java.util.Vector	1,019	25	43	185	106	44	24	4	1
java.util.zip.ZipFile	490	30	62	120	105	85	53	30	0
sparse	868	15	19	329	98	48	14	6	0
tsp	706	10	45	445	115	437	80	19	0
elevator	1,447	30	64	454	146	241	60	25	0
raytracer-fixed	1,915	10	50	565	200	105	39	26	2
sor-fixed	958	10	32	249	99	128	24	12	0
moldyn-fixed	1,352	10	39	983	130	657	37	30	0
Total	13,570	231	490	3,928	1,291	1,890	432	180	13
Total per KLOC		17	36	289	95	139	32	13	1

6.1 Non-Interference Specifications

Yield annotations provide a convenient and precise specification of exactly where thread interference may occur. We experimentally compare yield annotations with prior non-interference specifications based on atomic methods and on identifying race conditions. Specifically, we count the thread interference points in each benchmark under five different approaches for specifying non-interference as follows:

The ***NoSpec*** non-interference specification provides no information about thread interference, and so we must assume that interference might occur on any field access or any lock acquire, since these operations may conflict with operations of concurrent threads. We exclude operations that never cause interference, such as accesses to local variables, lock releases, method calls, etc. from this count.

The ***Race*** non-interference specification identifies fields with race conditions. Interference may occur only on accesses to these racy field or on lock acquires.

The ***Atomic*** non-interference specification identifies atomic methods. Thread interference may occur only in non-atomic methods, at field accesses, lock acquires, and also calls to an atomic method from a non-atomic context.

The ***AtomicRace*** non-interference specification identifies both race conditions and atomic methods. Thread interference may occur only in non-atomic methods, at racy accesses, lock acquires, and calls to atomic methods.

The ***Yield*** non-interference specification uses yield annotations to identify exactly those program points at which interference may occur.

Table 2 shows the number of interference points (IPs) in each benchmark under each non-interference specification. Benchmarks in which all methods are atomic, such as `Inflater`, have zero interference points under both the *Atomic* and *Yield* specifications. The results confirm that both *Race* (95 IP/KLOC) and *Atomic* (139 IP/KLOC) are useful non-interference specifications that significantly reduce the number of interference points in comparison to the base-case *NoSpec* (289 IP/KLOC). *AtomicRace* (32 IP/KLOC) combines complementary benefits from both *Atomic* and *Race*, and is significantly better than either of these approaches in isolation. Finally, *Yield* (13 IP/KLOC) provides a significant improvement over these prior approaches.

We sketch two situations illustrating why yield annotations, and the reasoning performed by our type system, are significantly more precise than *AtomicRace*. First, for the TSP algorithm from Figure 1, *AtomicRace* requires an interference point before each call to the atomic methods `path.isComplete()` and `path.children()` from within a non-atomic method. In contrast, our type system identifies that these two methods are more precisely characterized as both movers that do not interfere with other threads, and so no yield point is necessary at these calls.

As a second example, consider the method `append` from `StringBuffer` in Figure 5. This non-atomic method in turn calls two methods `sb.length()` and `sb.getChars()`, both of which are atomic (since the lock `sb` is not held at these call sites). Under *AtomicRace*, an interference point must be assumed before each of these calls to an atomic method from a non-atomic context. In contrast, our type system can verify that the lock acquire of `this` at the start of `append()` can move right to just before the `sb.length()` call, so no yield is required at the call to `sb.length()`. Thus, *AtomicRace* requires two interference points in this method, whereas our type system requires just one.

These two examples illustrate why the notions of race conditions and atomic methods are not by themselves sufficient to identify interference points in a precise manner, and the experimental results confirm that the cooperative type and effect system is significantly more precise that these prior approaches in its ability to verify interference points.

```
public final class StringBuffer ··· {
  (this ? mover : atomic) int length() { ... }
  (this ? mover : atomic) void getChars(...) { ... }

  compound synchronized
  StringBuffer append(StringBuffer sb) {
    ...
    int len = sb.length();
    int newcount = count + len;
    if (newcount > value.length) {
        expandCapacity(newcount);
    }
    sb..getChars(0, len, value, count);
    count = newcount;
    return this;
  }
}
```

Figure 5: StringBuffer

```
class RayTracerRunner implements Runnable {
  int id;

  compound public void run() {
    // init
    br.DoBarrier#(id);
    // render
    ..synchronized (scene) {
      for(int i=0;i<JGFRayTracerBench.nthreads;i++)
        if (id == i)
          JGFRayTracerBench..checksum1 =
              JGFRayTracerBench..checksum1
              + checksum;
    }
    br.DoBarrier#(id);
    // cleanup
  }
}
```

Figure 6: RayTracer

6.2 Examples of Defects

The primary purpose of yield annotations is to to facilitate formal and informal reasoning about higher-level correctness properties. At the same time, yield annotations also facilitate recognizing concurrency errors. To explore this aspect, the final column in Table 2 shows the number of *unintended yields* in each program, where we determined based on manual code inspection that thread interference is unintentional and erroneous. These unintended yields highlight concurrency bugs, including some atomicity violations and data races [13, 15] that could also be detected with prior tools.

StringBuffer and Vector. The yield annotation shown in Figure 5 highlights that other threads may concurrently modify `sb`, potentially causing `append()` to crash, in violation of its specified thread-safe behavior under JDK 1.4. A constructor in the `Vector` class suffers from a similar defect. Similar pitfalls occur in `Vector`'s inherited methods `removeAll(c)` and `retainAll(c)` [15]. In this experiment, we did not verify the correctness of inherited code, but JCC readily catches those errors when that code is checked.

RayTracer. The raytracer benchmark uses a barrier `br` to coordinate several rendering threads, as shown in Figure 6. After rendering, each thread acquires the lock `scene` before adding its local `checksum` to the global shared variable `checksum1`. However, each thread creates its own `scene` object, and thus acquiring `scene` fails to ensure mutual exclusion over the updates to `checksum1`. This is made clear by the explicit yields on the reads and writes of `checksum1`.[2]

SOR and Moldyn. In `sor` (see Figure 7), the computation threads synchronize on a barrier implemented as a shared two-dimensional array `sync`. Unfortunately, the barrier is broken, since the `volatile` keyword applies only to the array reference, not the array elements. Thus, the barrier synchronization code at the bottom of the main processing loop may not properly coordinate the threads, leading to races on the data array `G`. This problem is obvious when using JCC because racy annotations must be added in dozens of places, essentially to all accesses of `sync` and `G`. When the barrier is fixed, we obtain much cleaner code: the yield count decreases from 40 to 12. In particular, the accesses to `G` between barrier calls are free of yields, signifying that between barriers, sequential reasoning is applicable.

The `moldyn` benchmark uses a barrier object with a similar error in the use of `volatile`s and arrays. This bug leads to potential races on all data accesses intended to be synchronized by the barrier, and a large number (58) of yield annotations were necessary to document all such cases.

[2]We also note that if JCC were extended to identify locks used only by a single thread, we could remove the yield on the `synchronized` operation.

7. RELATED WORK

Cooperative multithreading is a thread execution model in which context switching between threads may occur only at `yield` statements [3, 4, 6]. That is, cooperative multithreading permits concurrency but disallows parallel execution of threads. In contrast, cooperability guarantees behavior equivalent to cooperative multithreading, but actually allows execution in a preemptive manner, enabling full use of modern multicore hardware.

Automatic mutual exclusion (AME) is an execution model ensuring mutual exclusion by default [22]; `yield` statements demarcate where thread interference is permitted. A key difference is that AME enforces serializability at run time via transactional memory techniques; in contrast, JCC guarantees serializability statically.

In prior work, we explored a simpler type system for cooperability [40], and dynamic analyses for checking cooperability and inferring yield annotations for legacy programs [41]. Others have explored *task types*, a data-centric approach to obtaining *pervasive atomicity* [24], a notion closely related to cooperability. Atomic sets are a useful, but complementary, technique for specifying groups of fields that must always be updated atomically. In contrast to cooperability, that approach enforces atomicity requirements by automatically inserting synchronization operations.

There is extensive literature on how to find and fix data races efficiently. Dynamic detectors may track the happens-before relation [14], implement the lockset algorithm [34], or combine both [28]. Static race detectors may make use of a

```
class SORRunner implements Runnable {
  double G[][];
  volatile long sync[][];

  compound public void run() {
    ...
    for (int p = 0; p < 2*num_iterations; p++) {
      for (int i=ilow+(p%2); i < iupper; i=i+2) {
        ...
        for (int j=1; j < Nm1; j=j+2){
          G[i][j]# = omega_over_four *
              ( G[i-1][j]# + G[i+1][j]# +
                G[i][j-1]# + G[i][j+1]# ) +
              one_minus_omega * G[i][j]#;
          ...
        }
      }

      sync[id][0]# = sync[id][0]# + 1;
      if (id > 0)
          while (sync[id-1][0]# < sync[id][0]#) ;
      if (id < JGFSORBench.nthreads-1)
          while (sync[id+1][0]# < sync[id][0]#) ;
    }
  }
}
```

Figure 7: Original SOR Algorithm

type system [7, 1], implement a static lockset algorithm [27, 30], or use model checking [31]. Data races often reflect problems in synchronization, and expose a weak memory model to programmers, compromising software reliability. Data race freedom remedies this issue by guaranteeing behavior equivalent to executing with sequentially consistent [2].

Atomicity is an analysis approach that checks if atomic blocks are serializable. Both static [15, 20, 37] and dynamic tools [12, 38, 16, 11] have been developed to check atomicity, as well as transactional memory techniques that enforce serializability at run time [21, 10, 35, 19]. While the notion of atomicity is very beneficial when reasoning about atomic methods, it is less helpful in documenting thread interference in non-atomic methods [40]. Moreover, atomicity introduces a form of bimodal reasoning, combining sequential reasoning inside atomic blocks with traditional multithreaded reasoning outside atomic blocks. In contrast, cooperative concurrency provides a uniform semantics for reasoning about the correctness of all code in a multithreaded system.

Deterministic parallelism generalizes atomicity to guarantee that the result of executing multiple threads is invariant across thread schedules. There are various approaches to ensure deterministic parallelism, including static analyses [5], dynamic analyses [32, 8], and run-time enforcement [29, 9].

8. SUMMARY

Reasoning about multithreaded software correctness is notoriously difficult under the preemptive semantics provided by multicore architectures. This paper proposes an approach where the programmer writes software with traditional synchronization idioms, and also explicitly documents intended sources of thread interference with yield annotations. Any annotated program verified by our type system behaves *as if* it is executing under a cooperative semantics where context switches between threads happen only at specified yield points.

This cooperative semantics provides a nicer foundation for reasoning about program behavior and correctness. In particular, intuitive sequential reasoning is now valid, except at yield annotations, and prior user studies have shown that yield annotations makes it significantly easier for programmers to identify defects during code reviews [33].

An important problem for future work is developing a type inference system that could infer the effect of each method, perhaps adapting ideas from earlier type inference algorithms for atomicity [15]. A related inference problem is identifying where to insert yield annotations into existing, unannotated program. Other interesting avenues for future work are to incorporate cooperative reasoning into a proof system, such as rely-guarantee reasoning, or to extend cooperability to reason about determinism properties.

9. ACKOWLEDGEMENTS

This work was supported by the NSF under grants CNS-0905650, CCF-1116883 and CCF-1116825.

10. REFERENCES

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[3] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Annual Technical Conference*, pages 289–302, 2002.

[4] R. M. Amadio and S. D. Zilio. Resource control for synchronous cooperative threads. In *International Conference on Concurrency Theory*, pages 68–82, 2004.

[5] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 97–116, 2009.

[6] G. Boudol. Fair cooperative multithreading. In *International Conference on Concurrency Theory*, pages 272–286, 2007.

[7] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 56–69, 2001.

[8] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *International Symposium on Foundations of Software Engineering*, pages 3–12, 2009.

[9] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2009.

[10] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, 2009.

[11] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Computer Aided Verification*, pages 52–65, 2008.

[12] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Symposium on Principles of Programming Languages*, pages 256–267, 2004.

[13] C. Flanagan and S. N. Freund. Adversarial memory for detecting destructive races. In *Conference on Programming Language Design and Implementation*, pages 244–254, 2010.

[14] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. *Commun. ACM*, 53(11):93–101, 2010.

[15] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *ACM Trans. Program. Lang. Syst.*, 30(4), 2008.

[16] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Conference on Programming Language Design and Implementation*, pages 293–303, 2008.

[17] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Symposium on Principles of Programming Languages*, pages 171–183, 1998.

[18] D. Grossman. Type-safe multithreading in Cyclone. In *Types in Language Design and Implementation*, pages 13–25, 2003.

[19] T. Harris and K. Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, 2003.

[20] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Conference on Verification, Model Checking, and Abstract Interpretation*, pages 175–190, 2004.

[21] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *International Symposium on Computer Architecture*, pages 289–300, 1993.

[22] M. Isard and A. Birrell. Automatic mutual exclusion. In *Workshop on Hot Topics in Operating Systems*, pages 1–6, 2007.

[23] Java Grande Forum. Java Grande benchmark suite. Available at `http://www.javagrande.org`, 2008.

[24] A. Kulkarni, Y. D. Liu, and S. F. Smith. Task types for pervasive atomicity. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 671–690, 2010.

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

[26] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Symposium on Principles of Programming Languages*, pages 378–391, 2005.

[27] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[28] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003.

[29] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97–108, 2009.

[30] P. Pratikakis, J. S. Foster, and M. Hicks. Context-sensitive correlation analysis for detecting races. In *Conference on Programming Language Design and Implementation*, pages 320–331, 2006.

[31] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *Conference on Programming Language Design and Implementation*, pages 14–24, 2004.

[32] C. Sadowski, S. N. Freund, and C. Flanagan. SingleTrack: A dynamic determinism checker for multithreaded programs. In *European Symposium on Programming*, pages 394–409, 2009.

[33] C. Sadowski and J. Yi. Applying usability studies to correctness conditions: A case study of cooperability. In *Onward! Workshop on Evaluation and Usability of Programming Languages and Tools*, 2010.

[34] 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, 1997.

[35] N. Shavit and D. Touitou. Software transactional memory. In *ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[36] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Conference on Programming Language Design and Implementation*, pages 115–128, 2003.

[37] C. von Praun and T. Gross. Static detection of atomicity violations in object-oriented programs. In *Workshop on Formal Techniques for Java-like Programs*, 2003.

[38] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32:93–110, 2006.

[39] J. Yi, T. Disney, S. N. Freund, and C. Flanagan. Types for precise thread interference. Technical Report UCSC-SOE-11-22, The University of California at Santa Cruz, 2011. At `http://www.soe.ucsc.edu/research/technical-reports/ucsc-soe-11-22`.

[40] J. Yi and C. Flanagan. Effects for cooperable and serializable threads. In *Types in Language Design and Implementation*, pages 3–14, 2010.

[41] J. Yi, C. Sadowski, and C. Flanagan. Cooperative reasoning for preemptive execution. In *Symposium on Principles and Practice of Parallel Programming*, pages 147–156, 2011.