Static Conflict Analysis for Multi-Threaded Object-Oriented Programs

Christoph von Praun and Thomas R. Gross
Laboratory for Software Technology
ETH Zürich
8092 Zürich, Switzerland

ABSTRACT

A compiler for multi-threaded object-oriented programs needs information about the sharing of objects for a variety of reasons: to implement optimizations, to issue warnings, to add instrumentation to detect access violations that occur at runtime. An Object Use Graph (OUG) statically captures accesses from different threads to objects. An OUG extends the Heap Shape Graph (HSG), which is a compile-time abstraction for runtime objects (nodes) and their reference relations (edges). An OUG specifies for a specific node in the HSG a partial order of events relevant to the corresponding runtime object(s). Relevant events include read and write access, object escape, thread start and join.

OUGs have been implemented in a Java compiler. Initial experience shows that OUGs are effective to identify object accesses that potentially conflict at runtime and isolate accesses that never cause a problem at runtime. The capabilities of OUGs are compared with an advanced program analysis that has been used for lock elimination. For the set of benchmarks investigated here, OUGs report only a fraction of shared objects as conflicting and reduce the number of compile-time reports in terms of allocation sites of conflicting objects by 28–92% (average 64%). For benchmarks of up to 30 KLOC, the time taken to construct OUGs is, with one exception, in the order of seconds.

The information collected in the OUG has been used to instrument Java programs with checks for object races. OUGs provide precise information about object sharing and static protection, so runtime instrumentation that checks those cases that cannot be disambiguated at compile-time is sparse, and the total runtime overhead of checking for object races is only 3–86% (average 47%).

Categories and Subject Descriptors

D.3.4 [**Software**]: Compilers; D.2.3 [**Software Engineering**]: Object-oriented programming—*Java*

This research was supported, in part, by a gift from the Microprocessor Research Lab (MRL) of Intel Corp.

PLDI'03, June 9–11, 2003, San Diego, California, USA.

General Terms

Program analysis, representations for concurrent programs

Keywords

Race detection, heap shape graph, object use graph

1. INTRODUCTION

A compiler for an object-oriented programming language with multi-threading needs precise information about the sharing of objects. The absence of precise information has undesirable consequences: the compiler must make many conservative assumptions, and they either inhibit a wide range of optimizations or lead to additional synchronizing operations.

A key issue is to determine the sharing of objects by threads. Previous work on escape analysis [2, 3, 6, 30, 21] has classified object accesses at compile-time according to properties of the accessed object. This information is stored in the global Heap Shape Graph (HSG) and provides the basis for improving the placement and kind of synchronization operations.

The HSG keeps information that is valid for all points in the program. If an object is accessed by two threads, then the object is considered to be shared. However, there are situations where more detailed analysis may reveal that the second thread starts only after the termination of the first, and in that case, no sharing of the object takes place. This paper presents a practical approach to analyse object-oriented programs to discover such cases. We construct an *Object Use Graph* (OUG) that approximates the happened-before relation [16] of access events that are issued by different threads to a specific object. The OUG augments the HSG and refines "escape" information: Instead of attributing a global classification to an object (or rather its compile-time abstraction) and the sites it is accessed from, the OUG recognizes structural, temporal, and lock-based protection of accesses in different lifephases of the object or contexts from which the object is accessed. This information is derived from the control flow inside individual threads and information about lock protection, object escape, thread-start and join. This information allows a variety of optimizations and, in addition, is precise enough to be useful for the static detection of synchronization defects.

The information in the OUG is the foundation for various applications in "concurrence-aware" compiler systems: (1) reporting of potentially conflicting accesses to the programmer; (2) sparse program instrumentations for dynamic

```
class Shared {
    int i;
    Shared () { i = 0; }                // (20)
}

class Example extends Thread {
    static Shared s_field;
    static Object lock_ = new Object();

    static void main(String[] args) {
        Shared s_local = new Shared();  // (2),(3)
        s_local.i++;                    // (4),(5)

        s_field = s_local;              // (6)
        s_field.i++;                    // (7),(8),(9)

        Thread t = new Example();
        t.start();                      // (10)

        synchronized(lock_) {
            s_field.i++;                // (11),(12),(13)
        }

        t.join();                       // (14)
        s_field.i++;                    // (15),(16),(17)
    }

    void run() {                        // (22)
        synchronized(lock_) {
            s_field.i++;                // (23),(24),(25)
        }                               // (26)
    }
}
```

Figure 1: Example Java program.

detection of access conflicts, e.g., [29, 7]; (3) compiling and optimizing programs in view of specific programming-level memory models, e.g., [18]; (4) optimization of synchronization and memory allocation in concurrent programs [2, 3, 6, 30, 21].

We implemented OUGs for standard Java programs and report here the evaluation for a set of multi-thread Java programs. The computation of OUGs requires a whole program analysis, and hence OUGs are designed for a way-ahead compilation and link model.

2. EXAMPLE

Figure 1 shows a simple program with two threads that both access an object of class `Shared`. We use this example to illustrate how OUGs differ from previous abstractions. The numbers in comments relate program statements to abstract events that are used during the analysis described in the following sections.

The terms *abstract object* and *abstract thread* refer to the compiler's entities that conservatively approximate runtime entities and do not necessarily correspond to unique runtime instances. The terms *runtime object* and *runtime thread* refer to actual objects and threads that exist when the program is executed. When there is no risk of confusion, we just talk about *objects* and *threads*.

2.1 Modeling of threads

A key aspect of OUGs is to explicitly distinguish the effect of different threads to abstract objects.

The Java language allows the compiler to determine the threads that may ever be created and started during a program run as well as the call-closure of methods executed by these threads. In addition to the initial thread starting at the `main` method, threads correspond to objects of class `Thread`. The type, entry method, and the multiplicity of threads are determined from the thread allocation sites (Section 3.1). The implicit invocation of class initializers does not generally allow to attribute their code to a particular runtime thread, and hence static initializers are modeled as separate initialization threads.

In the example of Figure 1, there is one `main` thread T_1 with entry method `Example::main`, one user thread T_2 with entry method `Example::run` and several init threads corresponding to class initializers.

2.2 Modeling of data

Java employs a simple memory model: Objects are allocated on a global heap and object access is possible only through references issued at object creation time. This model facilitates the computation of an approximation of the runtime object structure in the HSG at compile time. Nodes in the HSG represent individual runtime objects or sets of objects that are aliased. Edges represent points-to relations introduced through reference fields. The overall result of the shape analysis is a set of graphs rooted at class or thread nodes.

Figure 2 shows the HSG for the example program of Figure 1. Nodes in the HSG are given unique names and g_i refers to the node with unique id i. In this example, g_4 corresponds to class `Example` with its variables `s_field` and `lock_`. The node is marked as thread root, because it is the context of an init and the `main` thread. Similarly, nodes g_1, g_2, g_3 correspond to other classes. Nodes g_5 to g_{11} represent objects that are allocated during the execution of abstract threads in different contexts.

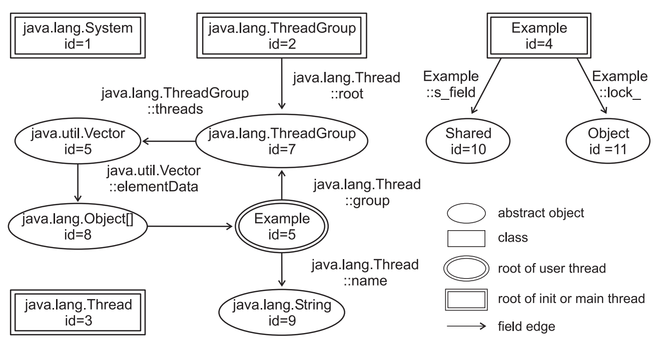

Figure 2: HSG for the example program.

2.3 Modeling of object uses

Object uses are modeled by OUGs. The nodes in the OUG represent *events*, edges represent a safe approximation of the happened-before relation. Events represent either program actions (accesses to fields, thread activity, ...), or record the compiler's analysis. Possible nodes in an OUG are:

GET/PUT: Read or write access to a field of the object.

LOAD/STORE/ESCAPE: Fetch or deposit a reference to the object from/to another variable. An ESCAPE node is a variant of a STORE node and occurs if the object holding the target variable is potentially shared among threads.

TSTART/TJOIN: Start or join of a thread.

ENTRY/EXIT: Thread entry and exit. These nodes do not correspond to a program action.

CALL: Method invocation site. These nodes are only used during the construction of the graph (Section 3); the effect of events that are issued downstream of calls are "inlined" into the graph at the position of the CALL node. Recursive calls sites are not unfolded, but connected to the surrounding invocation context (Section 3.3.3).

Nodes have a number of attributes: the abstract thread and the program site that issues the event, the thread that is managed (TSTART/TJOIN), the host object (LOAD/STORE/ESCAPE), the accessed field or method as well as the set of locks held during the access (GET/PUT/CALL).

There are three kinds of edges that express a general ordering of events for all program executions:

Control-flow ordering: Control-flow edges represent the order of events inside a thread. If a program accesses an abstract object inside a loop or recursion, the OUG of that abstract object is cyclic.

Reference-flow ordering: A thread cannot access an object before the creator thread of an object has made the reference available through a shared variable. This restriction is modeled by reference-flow edges that connect STORE/ESCAPE nodes with corresponding LOAD nodes. Reference-flow edges impose an ordering on events of the same thread or different threads.

Thread-relation ordering: A thread T cannot issue any event before T has been started. This fact is modeled by a TSTART node that precedes the ENTRY node representing the entry method of the started thread. Similarly, a TJOIN node follows the EXIT. These edges from/to TSTART and TJOIN are called thread-relation edges.

Two events in the OUG are *conflicting* if (1) there is no ordering between the events, and (2) the events stem from different threads, and (3) at least one event is a PUT, and (4) the accesses are not done under common lock protection. Events that are not conflicting with any other event are *safe*. This definition of a conflict is a compile-time abstraction for a *data race*. This approximation is safe because all runtime events that can constitute a data race are determined as conflicting in the OUG. As our compile-time view considers all control-flows possible, there might however be events that are determined as conflicting in the OUG that do not constitute a data race in any program execution.

Figure 3 shows the final OUG example of the abstract object g_{10} from Figure 3, and the construction steps are described in subsequent sections. Note that the analysis classifies the PUT and GET events (20), (4), and (5) as safe, because references to (objects represented by) g_{10} are not available to thread T_2 (they have not escaped at that stage of the execution). Events (8) and (9) are safe, because T_2 has not been started. Events (12), (13), (24), (25) are not ordered in the graph, there is however a common unique lock that is held during all accesses. Events (16) and (17) are again safe, because T_2 has been joined. Event (3) has been refined by the activities of the constructor (events (19), (20), (21)) and is therefore unlinked from the graph.

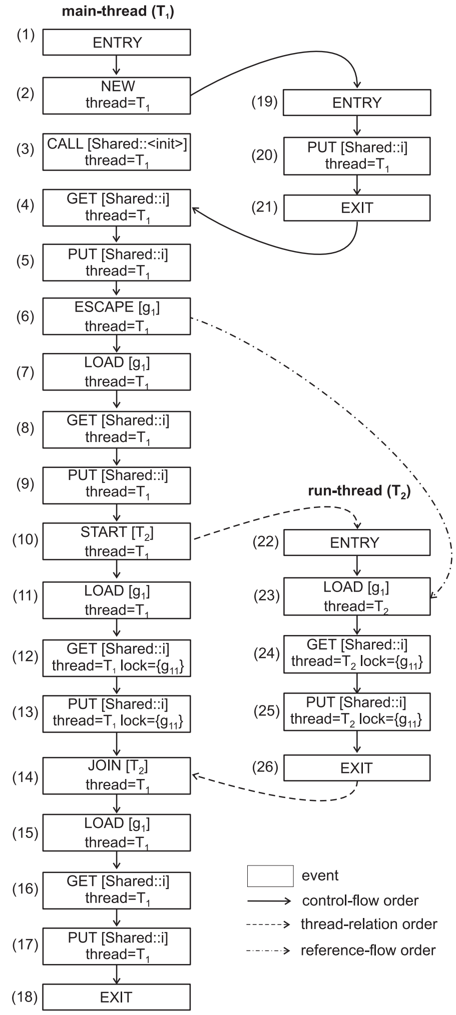

Figure 3: OUG for the abstract object g_{10} of Figure 2.

All update and read-/accesses to the `Shared` instance are hence ordered. The example demonstrates that the analysis of OUGs is able to detect different patterns that are commonly used to synchronize threads and protect shared objects from unordered access. Without detecting the temporal ordering, conservative assumptions would have to be made.

2.4 Thread-directed heap traversal

OUGs are built during a symbolic execution of the abstract threads. In an object-oriented program, the control flow of the execution follows a path through the nodes that represent the objects in the program. An action on a specific node (e.g., *getfield*, *putfield*) is noted as an event in the corresponding OUG. In that way, OUGs are built incrementally during the traversal of the HSG.

Figure 4 shows the HSG extended with temporary reference edges that correspond to reference relations through local variables along the execution of the `main` thread T_1. The heap traversal of T_1 is illustrated as a path along the reference edges with specific annotations at object accesses. Accesses correspond to method invocation or field access.

The initial sequence of the symbolic execution of the `main` thread is illustrated in Figure 4:

A. Start of thread T_1 at its entry method `Example::main`.

B. The object g_{10} of class `Shared` is created and initialized. Allocation and constructor invocation are noted as events (2) and (3) in the OUG of g_{10} (Figure 3). Events that are issued by the constructor of `Shape` are shown in a separate subgraph (events (19), (20), (21)). The subgraph is inlined into the the graph of the `main` method at the corresponding CALL event (3). For simplicity, this scenario omits the access to the superclass constructor in `Shape::<init>`.

C. Read and write of field `Shared::i`. The accesses appear as GET (4) and PUT event (5) in Figure 3.

D. Write access to field `Example::s_field` in g_4. Two events are generated in different OUGs: A PUT event is added to the OUG of g_4, and an ESCAPE event is recorded in the OUG of g_{10} (event (6) in Figure 3).

E. Read of field `Example::s_field` in g_4. Two events are generated in different OUGs: A GET node is added to the OUG of g_4 , and the LOAD event (7) is added to the graph that stands for the object pointed to by the reference that is being handled (OUG of g_{10}).

F. Read and write of field `Shared::i` in g_{10}. The accesses correspond to the events (8) and (9) in Figure 3.

G. Creation and initialization of g_5.

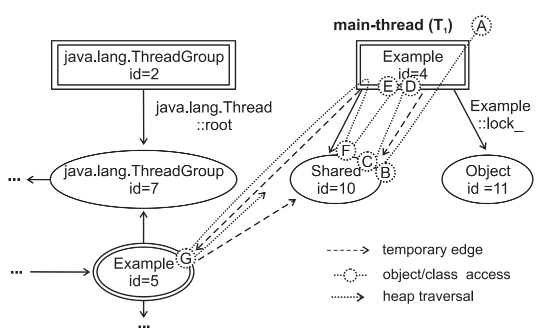

Figure 4: Fragment of the HSG with annotations of the start sequence of the heap traversal for the `Example::main` thread.

The Java language defines a synchronization mechanism that resembles Monitors [14], and locks are generally associated with objects. This mechanism enforces that the protection region of locks coincides with dynamic program scopes (methods or blocks), hence lock protection can be tracked along a symbolic execution and added as annotations to event nodes in the OUGs. Section 3.4 explains in detail how lock protection can be inferred from sets of locked abstract objects.

3. CONSTRUCTION OF OUGS

The overall process of constructing OUGs for a multi-threaded program follows several phases:

1. Determine the abstract threads in the program and their call graphs (Section 3.1).

2. Build the global heap shape graph (Section 3.2).

3. Build OUGs during a symbolic execution of the abstract threads (Section 3.3).

4. Analyze OUGs and determine conflicting events (Section 3.4).

The first and second phase are based on the procedures described in [21] and hence our description is brief.

3.1 Determining abstract threads

An *abstract thread* is a compile-time concept that represents a sequential control flow in a program. An abstract thread T is defined as

$$T := \langle tid, \langle m_0, ..., m_n \rangle, kind, multi \rangle$$

where *tid* is a unique id of the thread, and $m_0, ..., m_n$ specify the entry methods. Three *kind*s of abstract threads exist: an *init* thread for every class initializer method, one *main* thread corresponding to the unique entry point of the overall program, and *user* threads (instances of `java.lang.Thread` or its subclasses). *multi* specifies if multiple concurrent runtime instances of an abstract thread can exist. If a user thread has multiple or multiply executed allocation sites, then multiple instances are assumed to execute concurrently, otherwise abstract threads are *unique*. An allocation site is multiply executed, if it (or one of its callers) is inside a loop or recursion, or the allocating thread is not unique.

Each abstract thread is characterized by the methods it executes obtained from the call graph rooted at the thread entry methods. Our implementation bounds the entry methods for user threads as well the implementation alternatives at polymorphic call sites through a variable type analysis (VTA) [26].

3.2 Computing the HSG

The HSG represents a flow-insensitive model of global data and their reference relations, approximating object connectivity at any program point. Flow-insensitivity makes the HSG suitable for the analysis of multi-threaded programs, because the flow and progress of individual threads is generally not known at compile-time.

The HSG is computed according to Ruf's unification-based analysis [21]. It is *compositional* because every method is analyzed independently and method summaries are used to transfer the effect of method execution to individual call sites. A method summary is parameterized with caller context (parameters, return value(s), and thrown exception(s)), and hence the analysis is *context-sensitive*. The intra-procedural analysis is *flow-insensitive*.

The runtime values of reference variables are represented as *alias sets*. Alias sets stand for a runtime object or sets of such, and constitute the nodes in the HSG. In addition to abstractions for object instances, the HSG contains a node for each class modeling the class' variables.

An alias set AS is defined as

$$AS := \langle fieldmap, props, tidmask \rangle$$

where *fieldmap* maps fully qualified field names to alias sets representing the objects reachable through those fields. The flag *props* is used to note various properties, e.g., if the alias set is equivalent or connected to an alias set for a class or thread root (*global*) or if the object is accessed by multiple thread instances (*shared*). *tidmask* specifies the abstract threads that access the object(s) corresponding to the alias set.

The data structure supports the *union* of two alias sets, combining the field maps (the alias sets of the same fields are unified recursively) and the global flags (the result is global if at least one argument is marked global). Edges in the shape graph are featured by references to alias sets contained in the *fieldmap*.

The analysis associates variables with alias sets and unifies those alias sets in a stepwise process along the control flow of the program. Intra-procedural and inter-procedural analyses approximate the effect of method invocations.

3.2.1 Inter-procedural analysis

First, alias sets for class and abstract threads are created. These alias sets are the roots of the heap shape graph. Further alias sets are connected to these roots as a result of a sequence of intra-procedural steps (Section 3.2.2). Nodes that are transitively reachable through these root nodes are *global* and represent objects that are potentially accessed from multiple threads. The inter-procedural analysis considers all abstract threads and their methods in the reverse order of their invocation (bottom up traversal of strongly connected components in the call graph). Recursion is handled specially, for details, see [21].

3.2.2 Intra-procedural analysis

Given a method, the goal of the intra-procedural analysis is to establish a *method summary* that models the execution effect of a method to a calling context. A method summary captures aliases created through method invocation for data shared between caller and callee, i.e., parameters, return values, and thrown exceptions. In addition, the effect of the method execution in terms of allocated, read, and written objects is recorded. A method summary $MS[m]$ of a method m is defined as a tuple of alias sets

$$MS[m] := \langle\langle f_0, ..., f_n\rangle, ret, except, allocs, reads, writes\rangle$$

The computation of the method summaries starts with the creation of alias sets for all formal reference parameters $f_0, ..., f_n$ and local variables. *ret* and *except* are alias sets for the return values and thrown exceptions. Formal parameters are not aliased at this point, and caller-side aliasing is taken into account when a method summary is instantiated at call sites. A control-flow insensitive traversal of the statements of the method gradually builds the method summary. Assignment combines alias sets; field and array accesses create field-reference relations (all slots of an array are represented by a single symbolic field). Object allocation and access are recorded in the sets $allocs$, $reads$, and $writes$.

At a call site, the method summary of the callee is already available (see previous section). The summary is cloned and embedded into the method summary of the caller by unifying the alias sets of formal and actual parameters, return values and thrown exceptions. This instanced version of the callee's method summary is called a *method context*.

At object access sites (allocation, field/array access, method call), the id of the abstract thread is noted in the alias set representing the access target. After the creation of the HSG, this information is used to determine if objects are accessed by multiple or multiply executed abstract threads (hence *global* alias sets become *shared*).

3.3 Symbolic execution

The *symbolic execution* phase narrows the classification of abstract objects further and partitions shared abstract objects into *conflicting* and *non-conflicting*. For that purpose, OUGs are gradually constructed for all shared objects in the HSG. In an object-oriented environment, a symbolic execution maps nicely into a traversal of the heap shape structure determined in the previous analysis phase (Section 3.2). The current position of the traversal reflects the currently accessed abstract object.

3.3.1 Intra-procedural analysis

An OUG is assembled gradually during the symbolic execution, from individual *Method Object Use Graphs* (MOUG). Similar to an OUG at the whole program level, a MOUG models the relevant events at the level of a method m. A MOUG can be understood as a control-flow graph on which actions that do not result in events for the abstract object of interest are pruned.

$$MOUG[m, relevant] := \langle events, edges\rangle$$

The abstract object of interest is specified as a set of alias sets $relevant$. This set specifies either local alias sets $l_0, ..., l_n$ that, in a given method context, correspond to the object of interest, or as a global alias set if the object of interest corresponds to a class. Hence, for a specific method m, a number of MOUGs can exist, depending on the object of interest and aliasing that is given by the caller.

MOUGs are created in a single flow-sensitive method traversal, such that edges correspond to control-flow relations and nodes correspond to program actions. At any object allocation, access, and call site, the analysis determines if the statement is relevant for the abstract object of interest and consequently acts as follows: At allocation sites, a NEW node is created in the MOUG. At access sites to arrays or non-volatile, non-final fields, a PUT or GET is created. If a reference variable is handled that refers to the abstract object of interest, a STORE or LOAD event is created. At a call site, if the call target or one of the arguments to the call corresponds to the abstract object of interest, a CALL node is created. CALL nodes are not unfolded during the creation of MOUGs (see Section 3.3.2). Call sites of other methods can be considered as start and join as well if the callee starts a thread on any path, or joins a thread on all paths. We use a data flow analysis to determine these properties for all methods after the HSG is built.

Figure 5(a) shows an OUG for the `Example::main` method, where the object of interest is referenced by the first local variable (alias set l_1), holding the return value of the allocation statement of the `Shared` object. The local variable that refers to the lock of the synchronized block is l_5, the global alias set representing class `Example` is g_4. At the start of the symbolic execution of thread T_1 (or more precisely, at step B in Figure 4), the MOUG in Figure 5(a) is copied and mapped into the thread entry context, unifying alias sets l_1 with g_{10} and l_5 with g_{11}. Consequently, the STORE event (6) becomes an ESCAPE, because the target object g_4, which stores the reference to g_{10}, is shared. The CALL event (3) is unlinked, and a copy of the MOUG in

Figure 5(b) is inlined. The inlined nodes model events that the constructor `Shared::<init>` issues to the object that is being initialized (the `this` reference corresponds to alias set l_0). MOUGs are created on demand during the symbolic execution and cached. Figure 5(a) and (d) are both MOUGs for the `Example::main` method; they represent however a view on different sets of relevant objects, l_1 and l_2.

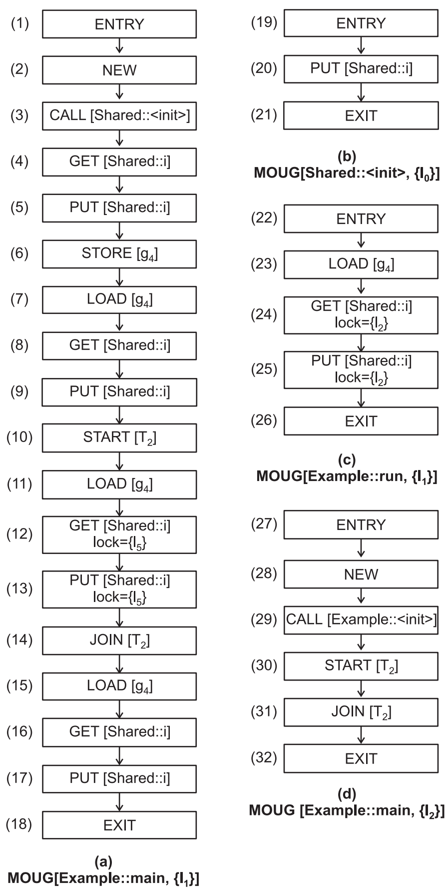

Figure 5: Example MOUGs.

3.3.2 Inter-procedural analysis

The analysis processes each abstract thread and its entry methods individually. The initial method context is determined from the thread-entry method and the thread root object. The analysis ignores the control-flow inside methods. During the traversal, the analysis keeps track of the abstract objects that are locked by the thread. The traversal of methods with block monitors is handled as a special case and follows the basic block structure such that the lock protection of individual statements is determined correctly.

When the symbolic execution of m encounters the first relevant event for an abstract object g, an initially empty OUG is created and associated with g. Before processing a statement, the analysis determines if an MOUG of method m in the current method context has already been mapped into the OUG of g. (The symbolic execution maintains a mapping between *method contexts* and the corresponding subgraphs. This mapping and the current statement allow the symbolic execution to determine the node in the OUG that corresponds to the current statement.) If not, the set L of local alias sets that correspond to g in the current method context is determined, then $MOUG[m, L]$ is computed (Section 3.3.1), cloned, and copied into the OUG of g.

The symbolic execution treats call sites as follows: First, the *method context* of the callee is determined (see Section 3.2.2). At this point, the method context is complete regarding the information about which alias sets are *global* or *shared*. The method context of the callee allows to determine a set S of *shared* objects that are read, written or allocated by the call. For each abstract object in $s \in S$, the appropriate MOUG of the callee is determined, considering aliasing at the call site (method context of the caller). This MOUG is inlined at the CALL node that corresponds to the current statement in the OUG of s. For polymorphic call sites, each target method is processed separately. At this point of the analysis, context sensitive type information associated with alias sets can be exploited to bound polymorphism. At a recursive call site, the symbolic execution reuses the method context of the corresponding active call and does not descend further. The reuse of method contexts during the symbolic execution is discussed in more detail in Section 3.3.3.

At a field or array access site, the current set of locks is attached to the access event in the affected OUG.

At allocation sites, a counter is incremented in the alias set that corresponds to the allocated objects. The increment accounts for the uniqueness of the allocating thread and whether the allocation or one of the call sites on the stack of the symbolic execution is in a loop or recursion. In contrast to the way uniqueness has been determined for threads (Section 3.1), this method of tracking the multiplicity of objects is context-sensitive and hence more precise. Information about the uniqueness of ordinary object is necessary to determine lock protection in the conflict analysis (Section 3.4).

After the symbolic execution, an OUG is a disjoint set of subgraphs resulting from different abstract threads. The events in the graphs are connected through control-flow edges. Then, reference-flow and thread-relation edges are added after the symbolic execution to create a coherent graph from the subgraphs. A reference-flow edge is added between STORE/ESCAPE and LOAD nodes that specify the same abstract object as host (and do not stem from the same method context). In addition, thread-relation edges are added between corresponding TSTART/ENTRY and EXIT/TJOIN nodes.

In the example program, the OUG of g_{10} in Figure 3 is combined from the MOUGs (a), (b) and (c) in Figure 5.

3.3.3 Optimizations

The symbolic execution can be the most expensive phase of the analysis because it considers all possible flows through the call graph of the program. If all call sites are followed in a straight forward manner, the worst case complexity of the symbolic execution is exponential in the number of call sites.

The first optimization is to avoid repeated descents into calls with equivalent method, thread and locking contexts. These three aspects of context in the symbolic execution are encoded in a *site context* SC:

$$SC := \langle m, \langle a_0, ..., a_n \rangle, tid, lockset \rangle$$

For the target method m of the call site, $a_0, ..., a_n$ are the actual parameters, *tid* is the id of the abstract thread that is processed, and *lockset* denotes the set of lock alias sets held at the call site.

The symbolic execution memorizes all processed call sites in terms of site contexts. At a call site, actual parameters from the current method context, *tid* and *lockset* are matched with site contexts of earlier invocations of m. In the matching, global alias sets are identified with their unique id, other alias sets can be determined as *fully local* if only local alias sets are reachable through their fields. A match means that the method *tid*, *lockset*, and all alias sets that are not fully local are equal; in that case, the symbolic execution does not descend into the call. This optimization resembles call caching in functional languages [13], although our mechanism for matching is simpler.

This first optimization has consequences that can deteriorate the precision of OUGs: The CALL node corresponding to a cached call site is not unfolded, but handled such that the subgraph of the earlier invocation is inlined. This procedure is safe with respect to the conflict analysis in Section 3.4, because the happened before relation is weakened: The current site, the earlier site and all events in between will be deemed to happen concurrently.

A second optimization is to avoid descents into methods that do not affect the state of shared data. At a call site the analysis determines from the method context if shared data is allocated, read or written. A generic form of this attribute is computed along the creation of method summaries (Section 3.2.2).

3.4 Conflict analysis

The *conflict analysis* determines conflicting events (as defined in Section 2.3) in an OUG. Runtime actions of conflicting events may participate in a data race. We assume that events issued by static initializers are generally not conflicting. This assumption allows to bypass conservatism about the concurrency of init threads (Section 2.1) due to the lack of information about implicit calls of static initializers. The assumption makes the analysis unsound, i.e., potentially conflicting accesses that involve static initializers are not reported. In practice however, we found that this assumption reduces the number of spurious reports and does not lead to underreporting for the programs we investigated (Section 5).

The conflict analysis identifies potentially conflicting PUT and GET events in an OUG of object g in four steps:

First, control flow and reference flow are considered: Events between a NEW and an ESCAPE event are safe because they happen before the object is accessible to threads other than the allocating thread, unless we detect that these events have a predecessor that is a successor to an ESCAPE.

The second step considers thread-relation ordering: Events that are not successors of TSTART are safe if all of them are issued by the same unique thread. Similarly, events dominated by a TJOIN are safe if they are all issued by the same unique thread. Two restrictions apply however: A thread might be started at several sites, and hence not all TSTART events of a thread may appear in the OUG of g (those that happen before g is allocated could have been omitted). In this case, TSTART is not a safe indicator for thread ordering. Moreover, a JOIN event is a safe thread relation information only if the joined thread is unique. In these cases, the TSTART or TJOIN nodes do not allow to infer protection properties from the ordering.

In the third step, the remaining access events are checked for lock protection. If only GET events remain, the object is classified as *readonly*. Else, an object g is *lock-protected* if the intersection I of locksets of all events is not empty and one of the following cases applies: (1) All accesses to g are performed through the `this`-reference and $g \in I$ is held. (2) g is object-local, i.e., only reachable through some hosting object h, and $h \in I$. (3) There is a unique lock object $u \in I$.

The fourth step is only done if lock protection cannot be determined at the object level. Then accesses are differentiated according to the fields they target, and lock or readonly protection is determined for individual fields as in the third step. If all fields are protected, the object is *mix-protected*, otherwise the object is classified as *conflicting*.

4. USE OF THE OUG

This section explains how the findings of the conflict analysis can be efficiently exploited in the executing program. The goal is to distinguish code that targets a conflicting object at runtime from code that targets safe objects. Hence the executed code should depend on the heap context of the execution, and different versions of the same method account for different classes of heap contexts (Section 4.1).

The result of this code classification is exploited by a program instrumentation that checks for object races during a program execution (Section 4.2).

4.1 Method specialization

In the view of OUGs, events originate at object access sites in the program code. Turning the view from objects to methods, the entirety of OUGs record the kinds of objects and the kinds of access that may happen at a specific object access site. The actual classification thereby depends on the heap context within which the method that performs the access is called. Specializations are created in 3 steps: (1) First, all heap contexts are identified within which a method operates. Every heap context provides a *specialization candidate*. This information is recorded during the symbolic execution. Then, the conflict analysis determines the conflict properties of abstract objects and individual access sites. (2) Then specialization candidates are classified and grouped according to the properties determined by the conflict analysis. (3) Finally, actual method specializations are determined and method invocation sites are adjusted to invoke specializations if necessary. In specialized methods, the resolution of polymorphic calls must consider not only the type/compile-time properties of the target object (as done with different variants of vtables), but also the calling context. Our implementation unfolds polymorphic call sites as cascades of `instanceof` checks, introducing additional runtime overhead.

4.2 Object race checker

We have developed a program instrumentation that detects *object races* [29] that occur at runtime. The system is based on the checking of locksets for accesses to objects that are actually shared. For the evaluation in Section 5.2, we use a simplified version of the system in [29] (no second ownership) that uses information from OUGs and optimizes the instrumentation through program transformation and data-flow analysis.

	philo	elevator	mtrt	sor	tsp	hedc	mold	ray	monte
program characteristics									
appl loc	81	528	11298	300	706	28299	1402	1972	3674
appl classes	2	5	34	7	4	48	11	19	19
lib classes	129	142	158	132	141	208	129	131	146
methods in call graph	192	311	722	205	302	1025	224	270	441
bytecodes in call graph	3605	6820	20137	4483	6481	24375	6531	5982	8161
user threads	2	2	2	3	2	5	2	2	2
method spec	68	118	578	16	108	3653	111	150	267
compilation resources									
shape analysis [s]	0.7	1.3	2.6	0.7	1.6	6.5	0.9	0.9	1.1
symb exec [s]	0.5	0.8	2.5	0.5	0.8	123.6	0.9	0.8	1.3
meth sites proc	103	191	1090	50	168	29254	156	209	431
meth sites reused	85	163	1640	43	123	60233	244	285	452
meth sites noeffect	100	179	855	81	163	29423	136	174	358
conflict analysis [s]	0.1	0.2	2.8	0.1	0.2	433.8	0.9	0.9	0.5
memory [MB]	0.5	3.0	14.7	0.5	1.5	263.5	1.5	3.4	3.8

Table 1: Benchmark characterization and compilation properties.

	philo	elevator	mtrt	sor	tsp	hedc	mol	ray	monte
classification of HSG nodes									
class	131	147	192	139	145	256	140	150	165
inst	43	65	199	44	62	467	51	71	79
inst unique	29	43	122	31	39	356	33	35	51
shared	10	13	97	3	13	184	16	29	36
shared readonly	3	6	55	1	6	116	6	12	28
shared lock-protected	6	3	36	1	4	30	6	6	2
shared mix-protected	0	0	1	0	0	2	0	3	1
shared conflicting	1	4	5	1	3	36	4	8	5
OUGs									
nodes max	217	327	1618	286	311	83052	537	302	726
nodes median	50	95	74	116	99	417	99	64	59
edges max	435	689	6083	410	640	206456	616	616	2450
edges median	67	172	111	221	163	748	145	92	84

Table 2: Characteristics of HSG and OUGs (no arrays).

5. EXPERIENCE

We have implemented OUGs in a Java-X86 way-ahead compilation environment. Our runtime system is based on GNU libgcj version 2.96 [12]. The numbers we present in the static and dynamic assessment refer to the overall program including library classes, and excluding native code. The effect of native code for aliasing and object access has been modeled explicitly in the compiler.

We use several multi-threaded benchmark programs [29] to evaluate the cost and precision of our program analysis (Section 5.1) and to quantify the runtime consequences on the example instrumentation for object race checking (Section 5.2.3).

philo is a simple Dining Philosopher application. elevator is a real-time discrete event simulator that is used as an example in a course on concurrent programming. Elevators are modeled as individual threads that poll directives from a central control board. Communication through the control board is synchronized through locks. The configuration we used simulates 4 elevators. mtrt is a multi-thread raytracer from the JVM98 benchmark suite [28], configured with 2 threads. sor (Successive Over-Relaxation over a 2D grid), and tsp (Traveling Salesman Problem) are data- and task-parallel applications with data access patterns of scientific codes; synchronization among threads is based on fork-join rather than locks. hedc is a warehouse for scientific astrophysics data developed at ETH [25]. This benchmark represents an application kernel that implements a meta crawler for searching multiple Internet archives in parallel. In the benchmark configuration, 4 principal threads issue random queries to 2 archives each. The individual queries are handled by reusable worker threads. The workload of this application kernel is typical for Internet server applications and similar to applications based on alternative mechanisms, such as Java Servlets. The programs mol(dyn), ray(tracer), monte(carlo) are multi-threaded numeric applications from the Java Grande benchmarks [15].

5.1 Compile-time characteristics

Table 1 describes the benchmarks and the results of the program analysis. The lines of code *appl loc* and classes *appl classes* account only for the application, not for the Java library. *lib classes* specifies the number of library classes that an application is linked with. The number of methods is given in *methods in call graph* including native and abstract ones, *bytecodes in call graph* specifies the size of the analyzed code.

The number of *user threads* specifies the number of abstract threads determined by the analysis, including the `main`-thread, not including init threads. Execution time and memory consumption of the compilation have been measured on a Pentium IV 1.4 GHz; the implementation of the analysis has not been tuned. *method spec* specifies the number of specializations that are generated for object race checking. *method spec* related to *methods in call graph* is an estimation for the code-bloat.

The shape analysis creates method summaries and processes each (non-native, non-abstract) method in the call graph once.

The symbolic execution phase is optimized to reuse results from earlier passes through methods if possible. The number *meth sites proc* specifies how many method invocation have actually been followed during this phase of the analysis. For all programs but hedc, the execution cost is on the order of a few seconds (and actually lower than, e.g., the cost to construct the call graph).

As discussed in Section 3.3, two optimizations are possible: *meth sites reused*, and *meth sites noeffect* list how often these optimizations have been effective. We see that these optimizations make the symbolic execution phase practical and result in more than linear gains of analysis time for larger programs (hedc, mtrt). The total saving is not just the numbers reported under *meth sites reused* and *meth sites noeffect*, but also includes nested calls that would have been followed if the analysis had descended and processed these calls. The long duration of the symbolic execution for hedc is due to the imprecision of available type and alias information. For this benchmark, 31 methods (some of which are frequently used in different contexts) are found in one strongly connected component of the call graph. Due to the loss of context sensitivity, spurious aliasing is created in the HSG among unrelated objects, leading to further imprecision and conservative assumptions in the downstream analyses. If objects that are actually local to methods or threads become global or shared due to aliasing, this means fewer optimization opportunities and hence more work during the symbolic execution.

Rows *class* and *inst* in Table 2 specify the number of nodes in the shape graph that correspond to classes or (sets of) object instances (not including arrays). *inst unique* is the number of unique objects of *inst*. Row *shared* quantifies the subset of nodes that are accessed from several (non-init) threads or a non-unique user thread during the symbolic execution. Shared nodes are further distinguished according to the results of the conflict analysis: Objects that are not written after they have escaped (determined by the OUG) are *shared readonly* (true for most alias sets that correspond to classes). Abstract objects with lock protection are reported as *shared lock-protected*. *shared conflict* specifies those abstract objects that experience conflicting accesses without lock protection on different fields. Object that use different protection mechanisms for individual fields, but have no conflicting fields, are given in row *shared mix-protected*.

Table 2 also reports the size of OUGs. For each program, we observe that there are a large number of small OUGs ($median \ll max$). Some OUGs are a lot larger than the median. As shown in Table 3, row *max alloc sites per obj*, there are some objects with a high degree of aliasing, and the OUGs of these objects are the larger ones. This property is crucial for hedc, which has 21 exceptionally large OUGs ($\#nodes + \#edges > 15000$). Our implementation of the conflict analysis only verifies lock protection in these cases, and classifies 9 objects as safe and 12 as conflicting.

Table 3 illustrates the precision of information in OUGs compared to other, more conservative variants for determining object sharing. Rows *global* specify global objects that are read and written by the main or user threads, and the total number of allocation and access sites. Such information could be reported by an escape analysis, e.g., [3, 6, 30]. Row *r/w shared* refines this set and only reports objects that are r/w accessed from multiple-threads as, e.g., determined by the analysis of Ruf [21]. In row *OUG (lock protection)*, the the OUG-based conflict analysis has been simplified (the first two steps in the conflict analysis in Section 3.4 are omitted and lock-protection is determined for all events in the graph). Finally the numbers for *OUG (all)* report the findings of the complete conflict analysis. The reduction compared to *r/w shared* quantifies the combined effect of considering (1) control-flow inside a thread, (2) the inter-thread relations, and (3) lock protection. The percentages given in row *improvement* specify the saving compared to *r/w shared*. We also specify the average and maximum number of allocation sites corresponding to NEW events in the OUGs per abstract object. This numbers indicates the degree of aliasing encountered, which is particularly high for hedc. In addition, the total number of *conflicting fields* and the average and maximum number of access sites corresponding to GET and PUT events per field are given.

The conflicts encountered by some abstract objects can be classified into four categories:

1. Row *all writes locked* specifies cases where all writes are lock-protected (and reads are not). Most of such conflicts could be identified as benign access pattern for lazy initialization (double-checked locking [23]). One of the reports in tsp corresponds to a global variable for the minimal tour length found so far. The updates are monotone and double checked, and concurrent reads of outdated information are tolerated by the algorithm. Hences this actual race is benign. Another report corresponds to objects that represent route information. In the actual execution, writes are ordered with respect to reads due to higher level synchronization, hence there is no actual race.

2. Row *object local to thread* specifies critical objects that are thread roots or object local to a thread root object. All conflicts reported in this category turned out to be benign, because the affected variables and objects are accessed by a single runtime thread These reports result from the inability of the analysis to distinguish different runtime instances of abstract threads that are not unique.

3. Row *one lock but not unique* specifies cases where the lockset is non-empty, but the analysis fails to determine the uniqueness of the locked object. In *elevator*, e.g., the locks protecting the data structures of individual floors are initialized in a loop and stored in an array. Despite the non-uniqueness of the lock in the compile-time view, the same lock instance consistently protects data for a certain floor at runtime. All reports in this category have been benign for the benchmarks.

4. The last category *no common lock* summarizes conflicts without common lock protection. In mtrt, a data race is found on the variable `RayTracer::threadCount`, which is however not relevant to the execution of the program. In mol, two reports refer to objects that are actually thread-local to their non-unique allocating thread. Another conflict report for this benchmark is benign, because the conflicting access is done only by one of multiple runtime threads with a specific id (control flow depends

on thread id). In ray, six reports correspond to objects that are initialized by the `main` thread and consequently associated with a specific instance of a worker thread that issues reads and writes. The conflict analysis recognizes that events of the `main` thread do not participate in a conflict; however, the worker thread is not unique and hence the analysis conservatively assumes that the read and write events by the worker threads conflict. For one reported object, a checksum is updated by the worker threads under common lock protection and read by the main thread after the worker threads are joined. The join is however not safely recognized, because the joined thread is not unique. In monte, a data race is found on a variable that flags if debug information should be printed. The race is benign, because the variable is always set to the same value. In hedc most reports are spurious for two reasons: Some critical objects are accessed from a non unique user thread; at runtime however, each instance is affiliated with only one actual thread. Moreover, some critical objects are accessed by different abstract threads without lock protection. At runtime, an ordering of these accesses is guaranteed by thread start and join, however the corresponding thread management events are not safely recognized by the conflict analysis. A true report points to an unsynchronized assignment of `null` to a shared variable `Task::thread_`, which could be read by another thread and lead to a `NullPointerException`.

When investigating the cause of conflicts, the reports list allocation and access sites for conflicting objects. For most programs, these lists are short, and the classification of the conflict allows easy analysis. For hedc, the scope of the inspection is enlarged due to aliasing.

5.2 Runtime aspects

5.2.1 Classification of objects

At runtime, we use a mechanism that precisely tracks accesses to objects from different threads including lockset information [29]. This runtime information together with the compile-time information tagged to the object headers allows to verify consistency between the results of the compile-time analysis and the actual runtime behavior. Objects that the runtime checker observed to be accessed by more than one thread are reported in Table 4 in row *actually shared*. The difference between *shared* and *actually shared* can be regarded as an indicator for the precision of our compile-time object classification.

There are two reasons for objects to be classified as *shared* that never become *actually shared* at runtime: First, there might be control-flow paths that enable actual sharing but they have not been taken in the specific program run we report. Hence in this case, the difference between *shared* and *non shared* is caused by an application property. Second, alias sets are conservatively classified as global/shared during the creation of the HSG. This phenomenon is especially critical for methods that are part of a recursion, where context sensitivity is lost (details in [21]).

The first aspect particularly affects tsp: Worker threads that determine and rate tours in the graph topology maintain route information in objects. Depending on the overall length of a tour, route information may be made available to other threads or may be dropped early by the thread that created it. The second aspect particularly affects hedc, where our current implementation makes conservative assumptions in the treatment of a large recursion (Section 5.1).

Row *actually conflict* lists the number of object races that have been determined by the runtime checker. This number is always lower or equal to *conflict*, and the difference is again an indicator for the precision of the static analysis. Some of the reported object races are not actual races since ordering is given through program properties, e.g., higher-level synchronization.

Some of the actual conflicts correspond however to real program defects (see static conflict detection in Section 5.1, and [29, 7]). For mtrt, tsp, and monte, the actual conflicts correspond to real data races and have been already determined by the static analysis (see Section 5.1). The reports for ray do not reflect real data races, due to an initialization before thread start ordering (see Section 5.1 why a static conflict has been assumed nevertheless). Similarly, the report on mol and most reports of hedc do not correspond to real races. In the latter, result data produced by worker threads is collected from containers after these threads have been joined, and hence accesses to the containers are naturally ordered.

5.2.2 Classification of object accesses

This section reports and quantifies the runtime benefit that is obtained from the precise classification of objects and access sites. Table 5 lists the dynamic number of field accesses according to a compile-time classification of objects. In row *stack-escape*, all accesses to stack-escaping objects are counted, similarly for categories *global* and *shared r/w*. The last two columns report accesses to objects that are identified as conflicting based on their OUG. For row *conflicting OUG flow-insensitive*, all field accesses to conflicting objects are counted. In row *conflicting OUG flow-sensitive*, only accesses through conflicting access sites are counted.

For most benchmarks, i.e., philo, elevator, mtrt, tsp, hedc, and monte, information from OUGs helps to reduce the number of accesses over *shared r/w*. Lock-protection, reference-flow, and inter-thread ordering contribute to the improvement. For some benchmarks, however, the analysis is not effective, mainly due to conservatism in the classification of objects and their access sites. In mol, e.g., most of the accesses target small objects that are solely accessed by the allocating thread. The OUG that corresponds to these objects is nevertheless found to be conflicting, because the objects are reachable through a field of the thread object (hence global), the accessing thread is not unique (hence shared), and the accesses do not happen under lock protection (hence conflict).

5.2.3 Object race checking

Table 6 reports the dynamic frequency of access checks in programs instrumented for object race detection (Section 4.2). The instrumentation associates a check with certain "critical" field and method accesses; not all "critical" field accesses need to be instrumented (details in [29]). An access check is done in two stages: First, an *inline* check tests if the accessed object is owned by the accessing thread; if so, execution can proceed. Otherwise, a *lockset* check verifies compliance with a certain locking policy. Rows *inline* and *lockset* in Table 6 report the corresponding numbers.

	philo	elevator	mtrt	sor	tsp	hedc	mol	ray	monte
global									
abstract objects	13	38	91	14	30	201	25	39	54
allocation sites	18	55	117	18	42	256	27	56	59
access sites	135	526	1002	288	478	1954	963	466	399
r/w shared									
abstract objects	7	10	59	5	9	107	14	24	20
allocation sites	12	17	89	4	13	180	19	43	29
access sites	111	246	956	197	337	1818	899	408	252
OUG (lock protection)									
abstract objects	2	7	8	5	6	76	8	18	20
allocation sites	2	9	19	4	5	163	7	31	29
access sites	21	168	165	155	190	1387	751	254	216
OUG (all)									
abstract objects	1	4	5	1	3	36	4	8	5
... improvement (%)	86	60	91	80	67	63	71	67	75
allocation sites	1	6	16	2	3	129	5	16	15
... improvement (%)	92	65	82	50	77	28	74	63	48
access sites	11	113	121	75	58	1110	529	144	118
... improvement (%)	90	54	87	62	83	38	41	65	53
avg/max alloc sites per obj.	1.0/1	1.5/2	4.0/9	2.0/2	1.0/1	4.1/63	1.3/2	2.0/4	3.8/9
conflicting fields	2	12	20	11	6	198	50	19	19
avg/max acc sites per field	5.5/8	9.3/29	5.7/23	6.8/11	9.7/14	4.8/33	10.6/127	5.5/13	5.9/23
conflict types									
all writes locked	0	2	1	0	2	11	0	0	2
object local to thread	1	1	1	1	1	2	1	1	0
one lock but not unique	0	1	2	0	0	8	0	0	2
no common lock	0	0	1	0	0	15	3	7	1

Table 3: Static conflict detection (no arrays).

	philo	elevator	mtrt	sor	tsp	hedc	mol	ray	monte
shared									
allocated	11	43	440	4	10011	861	2064	2103951	20020
actually shared	8	37	15	4	375	207	5	345	20013
conflict									
allocated	2	33	6	2	5002	491	2051	2103667	20007
actually conflict	0	0	1	0	163	15	1	69	1

Table 4: Allocation of objects with their compile-time classification and the actual situation at runtime.

The instrumentation has been applied with different criteria for determining "critical" objects. The number of dynamic inline checks decreases as the classification of "critical" objects becomes more precise (from row *stack-escape* to *OUG*). The magnitude of lockset checks follows this trend, the precise number may however vary for different program runs, depending on the thread schedule and the moment when objects become actually shared. A simple optimization of the instrumentation is to cover more than a single access with one check if accesses are not separated through thread synchronization. Rows *OUG optimized* report the resulting number of checks. For mtrt, hedc, mol and ray, the numbers are reduced significantly. In the case of mol, the static classification of object accesses has not been successful to reduce the number of critical field accesses (Table 5). Hence, depending on the kind of instrumentation, conventional optimizations can be an effective complement to the static analysis for reducing the runtime penalty of an instrumentation.

Table 7 quantifies the execution overhead for object race checking on a Pentium III, 933 MHz system. philo, elevator and hedc are not included since they are not CPU-bound; for the other benchmarks, we report the average time of three runs. The base version *orig* and the instrumented versions *stack-escape*, *global*, *shared r/w*, and *OUG* are compiled without optimization. The variants *optimized* and *OUG optimized* are compiled with loop transformation, and PRE for expression and redundant load elimination. The categories correspond to the those in Table 6.

The execution overhead of the instrumentation stems from lockset checking operations at certain critical object accesses. In addition, the resolution of polymorphic calls in the instrumented versions creates overhead (Section 4.1): Despite only about 1000 runtime checks (Table 6), mtrt is slowed down due to frequent calls of specialized methods (Section 4.1).

5.3 Limitations

The conflict analysis considers only accesses from user threads (Section 2.1). Initializer threads do not execute in separate runtime threads, but are invoked implicitly in the stream of *some* user thread. Hence object accesses from initializer threads can also participate in conflicts, and those conflicts are not detected by our procedure.

Our model does not consider the execution of *finalize* methods invoked from a separate *finalizer thread*. Consequently, our analysis of access conflicts does not take accesses in the scope of finalizer threads into account.

	philo	elevator	mtrt	sor$\times10^6$	tsp$\times10^6$	hedc	mol$\times10^6$	ray$\times10^6$	monte
stack-escape	8479	20957	231.3×10^6	150.6	696.4	149413	1311.4	3443.5	313.1×10^6
global	4783	17347	6.0×10^6	150.6	696.4	39808	1311.4	3391.1	651305
shared r/w	4728	17029	5.9×10^6	150.6	696.4	33480	1311.4	3391.1	490049
conflicting OUG flow-insensitive	136	10937	4578	150.1	521.5	27133	1310.0	2743.3	210017
conflicting OUG flow-sensitive	116	8662	3509	150.1	274.7	18144	1303.5	2740.7	210017

Table 5: Effect of the static analysis, number of field accesses to objects (no arrays).

	philo	elevator	mtrt	sor	tsp	hedc	mol$\times10^6$	ray$\times10^6$	monte
stack-escape									
inline	3610	5919	225.2×10^6	598	244.2×10^6	26177	637.9	2413.5	774040
lockset	1163	2559	5839	344	174.9×10^6	4764	316.5	326.4	195062
global									
inline	2218	5421	935123	572	244.2×10^6	12456	637.9	2377.3	340205
lockset	780	2548	2925	339	174.9×10^6	3294	316.5	326.4	160057
shared r/w									
inline	2257	5316	925564	509	244.2×10^6	11539	637.9	2377.3	340060
lockset	747	2527	816	303	174.9×10^6	3128	316.5	326.4	160019
OUG									
inline	3	2237	2331	5	69.2×10^6	7786	629.7	1941.5	40007
lockset	0	2201	2196	0	5519	1595	314.4	0.2	1
OUG optimized									
inline	2	2193	1038	4	56.2×10^6	6513	420.1	856.8	40007
lockset	0	2181	906	0	4971	902	104.8	104.6	2

Table 6: Effect of the optimization, number of dynamic checks done for object race checking (no arrays).

The approach of OUGs requires whole-program knowledge and hence Java features like reflection and dynamic class loading are not accommodated.

Information obtained from TSTART and TJOIN events is used conservatively (Section 3.4). Additional information about thread ordering, as could be obtained from annotations or user input, would improve the precision of this analysis (e.g., for hedc).

6. RELATED WORK

An exhaustive survey on problems and current research in the analysis of multi-threaded programs is given by Rinard [20].

Concurrency analysis aims at approximating the order of statements executed by different threads and computes the may happen in parallel (MHP) relation among statements. Statements are however not distinguished according to their execution context and the accessed data. The combination of MHP information with a model of program data (heap shape and reference information) could be used to determine conflicting data accesses. This approach is discussed by Midkiff et al. [18], but no compiler implementation results are yet available. OUGs naturally provide such an integration of control- and data-flow information in context-sensitive manner.

Bristow et al. [5] used an *inter-process precedence graph* for determining anomalies in programs with post-wait synchronization. Taylor [27], and Duesterwald and Soffa [9] extend this work and define a model for parallel tasks in Ada programs with rendez-vous synchronization. The program representation in [9] is modular and allows to efficiently analyze programs with procedures and recursion based on a data flow framework. Masticola and Ryder [17] generalize and improve the approach of [9] and provide experimental evidence of the effectiveness of their technique. Recent work from Naumovich et al. [19] computes the potential concurrency in Java programs at the level of statements. The authors have shown that the precision of their data-flow algorithm is optimal for most of the small applications that have been evaluated; medium to large sized benchmark programs have not been studied. The approach requires that the number of real threads in the system is specified as input to the analysis; the handling of recursion is not described in the paper.

OUGs borrow ideas from the program representations proposed in [5, 9]. However, OUGs partition the view on all program statements into sets of accesses statements to individual objects. This means a reduction in the size of the graphs which is crucial for conflict analyses with superlinear complexity. Moreover, OUGs account for an object-oriented model of memory, threads, and locks, tailoring the analysis of methods with regard to the heap-execution context (Section 3.3.2). Recent work of Choi et. al. [7] models concurrent threads in an *interthread call graph*, which is an extension of a call graph including edges for thread start but not for thread join. In addition, synchronized blocks are modeled explicitly by approximating locks held at call sites. The interthread callgraph contains one node per method and hence, unlike OUGs, does not distinguish method executions in different thread- and heap-contexts.

Sasha and Snir [24] have studied shared memory programs with *structured multi-threading* where the scope of parallelism is limited and known at compile time. The focus is determining inter-thread dependencies introduced through shared data access. They developed an analysis that inserts a minimum number of memory fence instructions into programs with data races, hence allowing sequentially consistent executions of such programs on weak memory hardware. OUGs can handle unstructured multi-threading as well.

In *object-oriented programs*, access to shared data is typically done indirectly through references. In such an environment, program analysis faces more difficulties to determine the relation between threads and the data they access

	mtrt	sor	tsp	mol	ray	monte
no instrumentation						
base	20.8	3.8	8.9	20.6	49.4	23.4
optimized[1]	19.9	3.2	8.9	—	46.1	22.6
object race checking						
stack-escape	41.7	3.9	23.8	64.0	116.1	41.5
global	29.6	3.9	23.8	65.5	111.5	41.3
shared r/w	29.0	3.9	23.9	65.4	110.9	42.0
OUG	28.5 (37%)	3.9 (3%)	10.3 (16%)	66.0 (220%)	82.7 (67%)	40.9 (75%)
OUG optimized	27.7 (39%)	3.3 (3%)	10.1 (14%)	38.4 (86%)	73.0 (58%)	40.4 (79%)

Table 7: Runtime in seconds and overhead of the program instrumentation (array access not instrumented).

than in languages with limited pointer usage. Flow-sensitive pointer analysis has been extended to multi-threaded programs by Rugina and Rinard [22]. Their algorithm for Cilk programs explicitly models the interference between parallel sections in the program and the additional aliasing created through this interference. Unlike Cilk, other object-oriented languages model concurrent activities themselves as objects, and hence the scope of concurrency is not limited to a static program scope (*unstructured concurrency*). A starting point for the optimization of such programs is to determine the locality of objects with respect to allocating threads. Based on different variants of pointer analyses, escape analyses have been developed for Java, e.g., [2, 3, 6, 30]. OUGs extend this work and model locking, threads- and their interaction with objects explicitly.

Other representations of parallel programs have been designed with specific optimizations in mind. Diniz and Rinard [8], e.g., focuses on the movement and elimination of lock-operations in automatically parallelized object-based programs. Their analysis is based on an inter-procedural control flow graph of all threads. This graph explicitly models the structure of locking and protected program scopes. Data are modeled as read and write sets that are attributed to the individual nodes of the graph. The transformations require that the program is free of data races, a condition met by automatically parallelized programs. Unlike OUGs, protection of data against concurrent access is only determined according to lock-protection and a known relation between locks and protected code regions. Polymorphism and different execution contexts of methods make it difficult to infer such a relation in typical object-oriented programs. OUGs, in contrast, compute a relation between data and their protecting locks.

All aforementioned approaches to relate threads and accessed data, including OUGs, are based on data- and control-flow information. Recent work on type systems has shown that data protection and locking policies can also be codified in data and method declarations that are checked statically. The main advantage of this approach is its modularity, which makes it, in contrast to a whole program analysis, well amenable to treat incomplete and large programs. Unlike OUGs, the application of these approaches to existing programs is not without difficulties: The type systems have either been proposed as extensions to existing programming languages [1, 4], or require annotations. Flanagan and Freund [10] present a type system that is able to specify and check lock-protection of individual variables. In combination with an annotation [11] generator, they applied the type checker to Java programs of up to 450 KLOC. The annotation generator is able to recognize common locking patterns and further uses heuristics to classify as benign certain accesses without lock protection. The heuristics are effective in reducing the number of spurious warnings; some are however unsound (but this property has not been a problem for the benchmarks investigated in [11]). OUGs model the reachability of objects explicitly and recognize cases of isolation beyond lock-protection that are covered by the heuristics automatically. The number of clustered warnings generated per KLOC is of the same magnitude as the reports we obtain for our benchmarks.

[1] mol is very sensitive to the cache layout of local variables and optimizations resulted in a performance degradation; for *OUG optimized*, loop transformations and PRE are disabled for this program.

7. CONCLUSION

The Object Use Graph is a concise extension of the Heap Shape Graph and provides a model of the runtime structure and the interaction of shared objects in the context of different threads. The OUG builds on previous work on pointer analysis and extends points-to information with information about the temporal relation of accesses. This information is thread-sensitive and approximates the complete set of possible accesses at runtime. We have implemented OUGs for a way-ahead compilation system for Java. A first evaluation for a set of non-trivial Java programs demonstrates that OUGs can be constructed with acceptable effort and that they produce tangible benefits for compilers and users.

OUGs provide information that is more precise than what can be obtained by analyses that do not model the temporal relationships. As a result, fewer accesses are approximately classified as conflicting, so a compiler that wants to draw the user's attention to those (possibly erroneous) accesses has fewer accesses to report. A runtime benefit can be realized as well if the compiler wants to insert dynamic checks to report actual sharing; there are fewer access sites to instrument.

As multi-threading is embraced by more users (and finds its way into future processor architectures), there will be increased demands on the compiler to provide reporting of access conflicts or to optimize placement and kinds of synchronization operations. OUGs are a solid foundation for concurrence-aware compilation systems.

8. ACKNOWLEDGMENTS

We thank Matteo Corti and Florian Schneider for their contributions to the compiler infrastructure and the referees for their detailed and insightful comments.

9. REFERENCES

[1] D. Bacon, R. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, pages 382–400, Oct. 2000.

[2] B. Blanchet. Escape analysis for object-oriented languages - Application to Java. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 20–34, Nov. 1999.

[3] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 35–46, Nov. 1999.

[4] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 211–230, Nov. 2002.

[5] G. Bristow, C. Dreay, B. Edwards, and W. Riddle. Anomaly detection in concurrent programs. In *Proc. Intl. Conf. on Software Engineering (ICSE'79)*, pages 265–273, 1979.

[6] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 1–19. ACM Press, Nov. 1999.

[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 *Conf. Programming Language Design and Implementation (PLDI'02)*, pages 258–269, June 2002.

[8] P. Diniz and M. Rinard. Synchronization transformation for parallel computing. In *Proc. Symp. Principles of Programming Languages (POPL'97)*, pages 187–200, Jan. 1997.

[9] E. Duesterwald and M. Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *Proc. Symp. Testing, Analysis and Verification (TAV4)*, pages 36–48, 1993.

[10] C. Flanagan and S. Freund. Type-based race detection for Java. In *Proc. Conf. Programming Language Design and Implementation (PLDI'00)*, pages 219–229, June 2000.

[11] C. Flanagan and S. Freund. Detecting race conditions in large programs. In *Proc. Workshop Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 90–96, June 18–19 2001.

[12] GNU Software. gcj - The GNU compiler for the Java programming language. http://gcc.gnu.org/java, 2000.

[13] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Proc. Conf. Programming Language Design and Implementation (PLDI'00)*, pages 311–320, June 18–21 2000.

[14] C. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, Oct. 1974.

[15] Java Grande Forum. Multi-threaded benchmark suite. http://www.epcc.ed.ac.uk/javagrande/, 1999.

[16] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. on Computers*, 46(7):779–782, July 1997.

[17] S. Masticola and B. Ryder. Non-concurrency analysis. In *Proc. Symp. Principles and Practice of Parallel Programming (PPoPP'93)*, pages 129–138, 1993.

[18] S. Midkiff, J. Lee, and D. Padua. A compiler for multiple memory models. In *Rec. 9th Workshop Compilers for Parallel Computers (CPC'01)*, June 2001.

[19] G. Naumovich, G. Avrunin, and L. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proc. 7th European Software Engineering Conf. and 7th Symp. Foundations of Software Engineering*, pages 338–354, Sept. 1999.

[20] M. Rinard. Analysis of multithreaded programs. In *Proc. Static Analysis Symp. (SAS'01)*, July 2001.

[21] E. Ruf. Effective synchronization removal for Java. In *Proc. Conf. Programming Language Design and Implementation (PLDI'00)*, pages 208–218, June 2000.

[22] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proc. Conf. Programming Language Design and Implementation (PLDI'99)*, pages 77–90, May 1–4, 1999.

[23] D. Schmidt and T. Harrison. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In F. Buschmann, R. Martin, and D. Riehle, editors, *Pattern Languages of Program Design (PLoP) 3*, pages 363–375, 1998.

[24] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. on Programming Languages and Systems*, 10(2):282–312, Apr. 1988.

[25] E. Stolte, C. von Praun, G. Alonso, and T. Gross. Scientific data repositories – designing for a moving target. In *Proc. Conf. ACM SIGMOD/PODS*, June 2003.

[26] V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, pages 264–280, Oct. 15–19 2000.

[27] R. Taylor. A general purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[28] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1996.

[29] C. von Praun and T. Gross. Object race detection. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 70–82, Oct. 2001.

[30] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 187–206, Nov. 1999.