Reflective Parallel Programming

Extensible and High-Level Control of Runtime, Compiler, and Application Interaction

Nicholas D. Matsakis
ETH Zurich
nmatsaki@inf.ethz.ch

Thomas R. Gross
ETH Zurich
trg@inf.ethz.ch

Abstract

Thread support in most languages is opaque and low-level. Primitives like wait and signal do not allow users to determine the relative ordering of statements in different threads in advance.

In this paper, we extend the reflection and meta-programming facilities of object-oriented languages to cover parallel program schedules. The user can then access objects representing the extant threads or other parallel tasks. These objects can be used to modify or query *happens before* relations, locks, and other high-level scheduling information. These high-level models enable users to design their own parallel abstractions, visualizers, safety checks, and other tools in ways that are not possible today.

We discuss one implementation of this technique, the intervals library, and show how the presence of a first-class, queryable program schedule allows us to support a flexible data race protection scheme. The scheme supports both static and dynamic checks and also permits users to define their own "pluggable" safety checks based on the reflective model of the program schedule.

1 Introduction

Although modern object-oriented languages like Java and C♯ have included support for threads from the beginning, the integration of threads into the runtime of these languages is fairly opaque. They provide no means, for example, to determine whether two threads acquire a common lock, or whether one thread joins another thread before executing. Partly this is because the runtime itself has very little information. The desired program schedule and relative timing for different threads is never stated explicitly by the programmer; rather, it is constructed imperatively via primitives like signals and barriers, and as a result there is no way for the runtime to know what it will be in advance.

In this paper we present a more explicit approach where programming languages include a high-level model for the program schedule, and they expose this model using first-class objects which support reflective queries and modifications. Programmers can then express their intended schedule in terms of the model, leaving low-level details to the runtime. This same model can also serve as the basis for other tools — such as visualizers, meta-programming frameworks, and safety checkers — developed by end users.

As a concrete example of the flexibility provided by such a scheme, we discuss the design of a user-extensible data-race detector. Checkers use mirrors [11] (reflective objects) of the program schedule to determine whether an access is permitted. Users can extend the system with their own checker implementations.

Our data race detector is designed around the *intervals library*. The intervals library models the program schedule using three concepts: *intervals*, which represent the span of time to complete a task; *points*, which represent individual moments during execution; and *locks*, which can be held by an interval for mutual exclusion. Users can specify and query the *happens before* relation between points as well as the locks which an interval will hold. The runtime is responsible for creating low-level threads and acquiring locks as needed.

The data race detector is based on *guard objects*, which are a generalization of locks. Every field and array element is associated with a particular guard. A guard object embodies a particular data race prevention scheme, which might be based on locks, the *happens before* relation, or something else entirely. Guard objects provide methods which can validate whether a given access is "safe" according to the guard's criteria. This judgement is generally derived from the abstract model of the program schedule exposed to them via reflection.

The system supports both static and dynamic checks, or even a mix of the two. Guards whose state is available at compile time can be checked statically. For

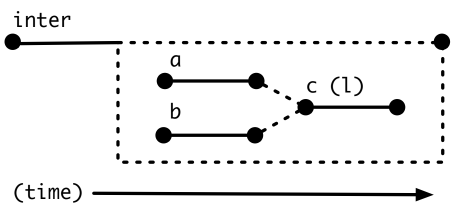

Figure 1: A sample interval diagram containing the interval `inter` and its three subintervals, `a`, `b`, and `c`.

those classes, dynamic checks are not needed, except for those locations where the compiler cannot prove static safety (in which case the checks must be provided explicitly, similar to a checked downcast). For other classes of guards, static checking is never possible. In those cases, the compiler will automatically insert safety checks wherever required.

The outline of the paper is as follows: we begin by briefly introducing the intervals model and showing how it can be manipulated and queried using our API. Next we outline the design for pluggable data race detection in more detail, and show how it makes use of schedule reflection. Finally we discuss related parallel models and other work before concluding the paper.

2 Constructing and Querying the Model

In this section we give a brief introduction to the intervals model. More in-depth introductions and examples, as well as a prototype implementation, can be found on our website [2] or in our previous work [16, 17].

Asynchronous tasks are represented by instances of the type `Interval`. Intervals are structured hierarchically and their parent is specified when they are created. Each interval executes in two phases. First, a method `run()` is executed. If the interval's task is very simple, then `run()` may simply perform it in its entirety. Otherwise, it can create a number of parallel subintervals. The second phase executes these subintervals in parallel with one another once the `run()` method has returned.

Each `Interval` has two associated points, `start` and `end`, which are instances of the type `Point`. The start point represents the moment when the interval begins to execute, and the end point represents the moment when the interval and all its subintervals complete. Points are related through a *happens before* relation.

Mutual exclusion is supported via instances of the type `Lock`. Every interval can be associated with one or more locks, which are automatically acquired before its start point occurs and released after its end point has occurred.

The program schedule is made up of all the instances of these three types and their interconnections. We depict

```
 1  void method(Interval parent, Lock l) {
 2      Interval inter = interval (parent) {
 3          Interval a = interval { ... };
 4          Interval b = interval { ... };
 5          Interval c = interval { ... };
 6
 7          a.end.addHb(c.start);
 8          b.end.addHb(c.start);
 9          c.addLock(l);
10
11          assert b.end.hb(c.start);
12          assert c.locks(l);
13      }
14  }
```

Figure 2: Sample code to create the schedule shown in Figure 1.

schedules graphically using *interval diagrams*. Figure 1 gives an example containing one parent interval, `inter`, with three leaf subintervals, `a`, `b`, and `c`. The solid circles represent points. The solid lines beginning at an interval's start point represent the period of time when the interval's `run()` method is executing. The dashed box indicates the period where the interval's subintervals are executing in parallel (if an interval has no subintervals, the dashed box is omitted). Dashed lines between points, such as those connecting the end of `a` and `b` to the start of `c`, indicate *happens before* edges specified by the user. The `(l)` in the label of subinterval `c` indicates that `c` executes while holding the lock `l`.

Figure 2 provides an example of code to create the schedule from Figure 1. The code is presented in a Java-like language extended with a keyword `interval` which creates a new interval with the given run method.[1]

The method takes two parameters: an existing interval `parent` (not shown in the diagram) and a lock `l`. It creates the interval `inter` on line 2 with `parent` as its superinterval. When `inter` is scheduled, its `run()` method will then in turn create the intervals `a`, `b`, and `c` on lines 3–5 (because no super interval is specified, the default is the current interval, or `inter`).

The methods `addHb()` and `addLock()`, shown on lines 7–9, are used to relate existing objects. $p1$`.addHb(`$p2$`)` creates a new *happens before* edge between $p1$ and $p2$, and i`.addLock(`l`)` ensures that interval i will acquire lock l.

Similarly, the methods `hb()` and `locks()`, shown on lines 11 and 12, can be used to query the schedule. $p1$`.hb(`$p2$`)` returns `true` if $p1$ *happens before* $p2$, and i`.locks(`l`)` returns `true` if the interval i will hold lock l when it executes.

[1] In the library implementation, this keyword is replaced with a Java anonymous class.

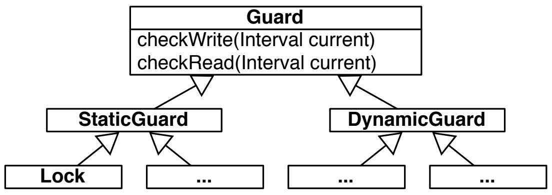

Figure 3: UML diagram showing the `Guard` hierarchy.

3 Pluggable Data Race Detection

Our data race detector builds on recent type systems [8, 4, 13] that use annotations to associate each field with a lock. We generalize the lock for a field into a *guard object*, which defines methods that check whether the field can be safely read or written. A single guard may protect fields belonging to multiple objects, and a single object can make use of multiple guards. This scheme gives users complete control over both the granularity of the data race detector — which helps to avoid high-level data races [9] — and the degree of data race freedom required.

All guard objects must be derived from the type Guard shown in Figure 3. The runtime and compiler can use the two `check` methods shown to determine whether a particular access is permitted. Both methods work the same way: If it is safe for the interval `current` to perform the given kind of access, the method returns normally. Otherwise, it throws an exception: the precise exception is not specified, but it should contain details that help the user to debug the problem.

The two subtypes of `Guard` distinguish guards that should be checked at compilation time from those that must be checked dynamically. These subtypes will be discussed in more detail shortly.

All guards must ensure that once an interval is permitted to read or write, all later checks of the same kind by that interval will succeed. This requirement implies that although every read or write access must be preceded by a check method, there is no need to check twice within a single interval. It also implies that checks and accesses do not need to be performed atomically. Of course, to be useful, the various `check` methods should also enforce some coherent data race protection scheme, but this is not strictly required by the API.

3.1 Statically Checked Guards

Subtyping `StaticGuard` is a signal to the compiler that the correctness of the guard is intended to be checked statically. Such guards have certain special requirements that enable them to be used within the compiler. First, their `check` methods must not rely on external state or objects which are not available at compile time. Second,

```
class Lock implements StaticGuard {

    void checkRead(Interval current) {
        checkWrite(current);
    }

    void checkWrite(Interval current) {
        if(!current.locks(this))
            throw new LockRequired(
                current, this);
    }

}
```

Figure 4: The methods on the `Lock` class which allow it to be used as a guard. The lock must be held by an interval for it to perform any kind of access.

their check methods must be monotonic, meaning that if a given schedule is judged race free, adding additional *happens before* relations or locks will not cause the guard to report a race. This is needed because the compiler has only a partial view of the schedule at any given time.

Whenever the compiler finds an access to a field whose guard is a subtype of `StaticGuard`, it will invoke the guard's methods with whatever subset of the schedule it can derive from the method containing the access (more on this below). If this check succeeds, then the compiler knows that the same check would succeed at runtime, and the access is permitted.

If the check fails, a compile-time error is generated. If possible, users should adjust the program as needed to make the check succeed. In some cases, however, the type system may simply not be strong enough to show that the access is safe, even though the user knows for certain that it is. In that case, users can insert an explicit invocation of the check method; this serves as an assertion that the access is indeed safe. If the assertion is false, then the check method will throw an exception at runtime.

An example of a statically enforcable guard is a `Lock`. The source for such a guard is shown in Figure 4. The check methods use the reflective method `locks()` to check that the current interval holds the lock. This check is monotonic, as it will not fail if the interval should hold other locks.

We will now demonstrate how the compiler checks guards statically, using the method `static()` in Figure 5 as an example. The method `static()` creates an interval `x` which will write the field `someField`. As `someField` is guarded by `someLock`, this access is only safe if `someLock` is held. Therefore, line 12 states that interval `x` should acquire the lock `someLock`.

When the compiler checks this method, it is able to

```
1  class SomeClass {
2
3      final Lock someLock;
4
5      @GuardedBy(someLock)
6      String someField;
7
8      void static(Interval parent) {
9          Interval x = interval (parent) {
10             someField = "...";
11         }
12         x.addLock(someLock);
13     }
14
15     void dynamic() {
16         someLock.checkWritable();
17         someField = "..."
18     }
19
20 }
```

Figure 5: In the first method, the compiler can ascertain statically that `someLock` is held when `someField` is modified. In the second method, however, a dynamic check is required.

derive the schedule shown in Figure 6(a), which shows that `x` is a subinterval of `parent` and that it will hold `someLock` when executing. It therefore constructs a set of mirrors — essentially, mock objects implementing the same interface as real intervals, points, and locks — for that known schedule, as shown in Figure 6(b). It then instantiates the guard for `someField` (which in this case is actually the mocked up version of `someLock`) and executes its check method. If this check method succeeds, the access is considered safe.

Should the check method fail, however, the user can always get the program to compile by manually inserting an invocation of the check method. This case is shown in the method `dynamic()` on line 16 of Figure 5.

3.2 Dynamically Checked Guards

The type `DynamicGuard` is a supertype for guards whose correctness is not intended to be checked statically. Such guards usually maintain state during execution (such as the interval which last wrote to a field) or are otherwise dependent on objects that only exist at runtime. In this case, all accesses to their protected fields must be preceded by an invocation of the appropriate `check` method. To spare users the tedious task of inserting these checks by hand, the compiler will automatically insert checks where needed for any guard whose type descends from `DynamicGuard`.

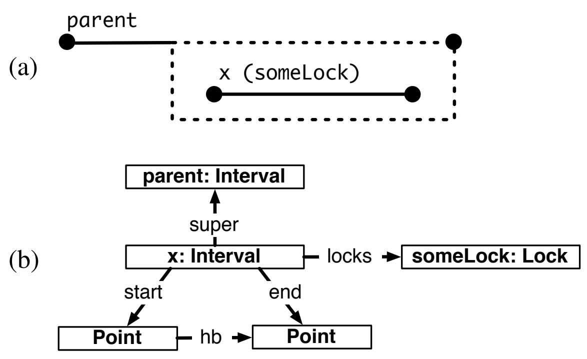

Figure 6: (a) Interval diagram and (b) mirrors derived by the compiler for the method `static1` from Figure 5.

3.3 Intentionally Racy Guards

It often happens that a particular field can tolerate a certain amount of “raciness” without affecting the correctness of the algorithm. For example, branch-and-bound algorithms track upper- and lower-bounds as they search. When a particular set of data is shown to fall outside of these bounds, it can be rejected. However, acquiring a lock merely to read the current bounds is often too expensive to perform regularly; since the bounds are just a hint in any case, it is acceptable to read them in a racy fashion and only guarantee that updates are strongly ordered.

This case can be easily accomodated by defining an appropriate guard which requires a lock for writes but not reads. Using a distinctive guard also makes it easy to identify those parts of the program which permit data races for special auditing or code review.

3.4 Practical Considerations

As the focus of this paper is not the data race check itself but rather its use of reflection, we have omitted discussion of many features that can lighten the annotation burden and dynamic overhead. The full system ensures that most classes can remain ignorant of guards or other synchronization mechanisms.

4 Related Work

The *happens before* relation was originated by Lamport [14] and figures prominently in the Java memory model [15]. Using *happens before* relations as the basis for the high-level schedule ensures a one-to-one correspondence between the memory model and scheduling constraints, thus avoiding the need for users to relate two separate concepts.

The technique of using mirrors at compilation time to describe a subset of the program schedule is inspired by `javac`'s annotation processing API [3], which uses mirrors to expose a subset of the class hierarchy that will exist at runtime.

The concurrent object-oriented language ABCL/R had reflective capabilities [21], but they were quite different from those described in this paper. Rather than abstractions for describing and controlling the program schedule, ABCL/R provided means for users to monitor and affect in-flight messages between distributed objects.

Smalltalk and its variants support reflection over the program stack. This allows Smalltalk to support debuggers and a wide variety of meta-tools without recourse to an alternate API. However, Smalltalk's parallel programming model lacked a high-level model for the schedule.

Race detection and prevention schemes, both static and dynamic, have a long history in the literature. The closest system to the static checks we describe in this paper is `rccjava` [8], which checks that fields are only accessed under appropriate locks. Our system is a generalization of theirs to accomodate multiple field protection schemes as well as checked dynamic assertions.

The use of guards for grouping related fields is similar to data coloring [12] or atomic field sets [20], two approaches which aim to eliminate high-level races through a data-centric approach. One important difference, however, is that guards in our system correspond to real objects and are not erased at runtime.

Imperative languages like C and Java have seen many language extensions [18, 7, 10] and libraries [1, 6, 5, 19] that aim to provide higher-level models for parallelism, generally layered on top of threads. Although some of these systems offer first-class objects that represent tasks, they do not provide any sort of high-level reflective interface, nor do they allow users to specify their schedule in terms of the model directly.

5 Conclusion

The presence of a first-class, queryable model for the program schedule opens the door for a new kind of meta-programming, one based on the program's schedule rather than its type structure. The model serves as a flexible, high-level interface between the application and the runtime, allowing the application to both constrain and query the parallel control flow.

In this paper we have focused on one application of reflection (customizable data race detection) with one model (the intervals model). However, the idea can also be applied to other potential uses (visualizers, deadlock analysis, etc) as well as to models derived from other parallel languages, such as Cilk [18] or OpenMP [7].

References

[1] Concurrency JSR 166 Interest Site. `http://gee.cs.oswego.edu/dl/concurrency-interest/`.

[2] Intervals Web Site. `http://intervals.inf.ethz.ch/`.

[3] JSR 175. `http://www.jcp.org/en/jsr/detail?id=175`.

[4] JSR 308. `http://pag.csail.mit.edu/jsr308/`.

[5] Managing Concurrency with NSOperation. `http://developer.apple.com/Cocoa/managingconcurrency.html`.

[6] Parallel Extensions to the .NET Framework. `http://msdn.microsoft.com/en-us/concurrency/default.aspx`.

[7] OpenMP Specification 3.0. `http://openmp.org/`, May 2008.

[8] Abadi, M., Flanagan, C., and Freund, S. N. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst. 28*, 2 (2006).

[9] Artho, C., Havelund, K., and Biere, A. High-level data races. *Software Testing, Verification and Reliability 13*, 4 (2003).

[10] Bocchino, Jr., R. L., Adve, V., Adve, S., and Snir, M. Parallel Programming Must Be Deterministic by Default. In *First USENIX Workshop on Hot Topics in Parallelism (HotPar)* (2009).

[11] Bracha, G., and Ungar, D. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA* (2004), ACM.

[12] Ceze, L., von Praun, C., Caşcaval, C., Montesinos, P., and Torrellas, J. Concurrency control with data coloring. In *MSPC* (2008), ACM.

[13] Grossman, D. Type-safe Multithreading in Cyclone. In *TLDI* (2003), ACM.

[14] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978).

[15] Manson, J., Pugh, W., and Adve, S. V. The java memory model. In *POPL* (2005), ACM.

[16] Matsakis, N. D., and Gross, T. R. Programming with Intervals. In *LCPC* (2009), Springer.

[17] Matsakis, N. D., and Gross, T. R. Handling Errors in Parallel Programs Based on *Happens Before* Relations. In *HIPS (to appear)* (2010).

[18] Randall, K. H. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Dept. of EECS, MIT, May 1998.

[19] Reinders, J. *Intel threading building blocks*. O'Reilly & Associates, Inc., 2007.

[20] Vaziri, M., Tip, F., and Dolby, J. Associating synchronization constraints with data in an object-oriented language. In *POPL* (2006), ACM.

[21] Watanabe, T., and Yonezawa, A. Reflection in an object-oriented concurrent language. In *OOPSLA* (1988), ACM.