

Thread Safety Through Partitions and Effect Agreements

Nicholas D. Matsakis and Thomas R. Gross

ETH Zurich

Abstract. This paper describes a safety analysis for a multithreaded system based upon transactional memory. The analysis guarantees that shared data is always read and written from within a transaction, while allowing for unsynchronized access to thread-local and (shared) read-only data, as well as the migration of data between threads. The analysis is based on a type and effect system for object-oriented programs called *partitions*. Programmers specify a partitioning of the heap into disjoint regions at a field-level granularity, and then use this partitioning to enforce safety properties in their programs. Our flow-sensitive effect system requires methods to disclose which partitions of the heap they will read or write, and also allows them to specify an *effect agreement* which can be used to limit the conditions in which a method can be called.

1 Introduction

The Java language’s traditional, lock-based model of parallel programming has proven to be vulnerable to a number of serious problems, such as deadlock and priority inversion. Programmers are forced into a difficult trade-off between correctness and performance. While a coarse-grained locking scheme is the easiest to implement and verify, fine-grained locking – or even nonblocking algorithms – is required to take full advantage of the parallel hardware.

Recent research in transactional memory [1] offers an attractive alternative. Programmers can use `atomic` statements to group together operations which must – for reasons of correctness – be executed without interruption, and the runtime system will ensure efficient and correct execution.

As convenient as transactional memory is, it is not a panacea. One obvious problem that programmers will face is verifying that they use `atomic` sections consistently. For example, shared objects should only be modified within a transaction, while thread-local and read-only objects may be used freely. Because mainstream programming languages today offer no means to controlling aliasing, verifying that each object is used safely must be done manually and is error-prone.

In this paper, we present a technique that guarantees consistent usage of transactions within a program. Our work is done in the context of a Java-like language augmented with transactional memory. It supports a number of common threading patterns, including thread-local data, read-only data, and the transfer of objects between threads.

Our technique is based on two new language constructs:

1. *Partitions* extend the type system so that it can describe aliasing at a very fine-grained level. Programmers use partitions to expose the aliasing structure of their program. A flow-sensitive effect system then is used to determine which partitions may be read or written by the program at different points in time.
2. *Effect agreements* allow a method to constrain the effects of its caller, while preserving modular compilation. For example, a method which hands off data in a particular partition to a new thread may prevent its caller from modifying that data while the new thread is executing.

The paper begins with an introduction to partitions and discusses how they are integrated into the type system. We then discuss the effect system, and in particular effect agreements, and give a brief example which uses them to verify the thread safety of a simple web server. Finally, we present the algorithm used to check that effect agreements are respected and close by showing how to extend the `Thread` class so as to enforce safe threading practices throughout the program.

2 Partitions at a Glance

A *partition* is a compile-time abstraction that describes a portion of the heap at a field-level granularity. In other words, if we define the heap $\mathcal{H}(o, f) \mapsto o$ as a mapping from an (object id, field) pairs to another object id, a partition is a set of such (object id, field) pairs. Partitions are similar to memory regions, except that we do not use them for memory management, but rather for alias tracking.

In our system, class and method definitions are parameterized by a set of *partition parameters*, analogous to generic type variables. When the class is instantiated or the method is invoked, each partition parameter will be mapped to a fixed partition.

2.1 Code Example

To demonstrate how partitions are integrated into the language, Figure 1 gives the definition of a class `IntWrapper`. `IntWrapper` has a single partition parameter, named `P`, which is indicated by the `@P` which follows the class name. It contains a single field `field` located in the `P` partition, and two accessors `get()` and `set()`. The partition of `field` is indicated by the `@P` preceding its type.

The `clone()` method of Figure 1 demonstrates many important points:

1. Methods can have partition parameters in addition to classes. In this case, the values for these parameters are inferred when the method is invoked.
2. New partitions can be created via a statement like `new @R`, where the name `R` of the fresh partition should not shadow any other partitions in scope.

```

class IntWrapper @P {
    // Leading @P indicates this field is in partition P.
    @P int field;

    // Primitive types require no partitions.
    int get() { return field; }
    void set(int i) { field = i; }

    // Methods can be parameterized with partition variables too.
    @Q IntWrapper@Q clone() {
        new @R; // demonstrate syntax to create a new partition
        IntWrapper@Q res = new IntWrapper@Q();
        res.set(field);
        return res;
    }
}

```

Fig. 1. Introductory example showing how partitions are integrated into the syntax.

- When new instances of a class are created, concrete values must be given for each partition parameter of the class. In this case, the expression `new IntWrapper@Q()` creates an `IntWrapper` instance whose `P` partition parameter is bound to `Q`, the partition parameter of the `clone()` method.

Although this example does not take advantage of it, it is also possible to take the union of several partition parameters. For example, the `new` statement in the `clone()` method could have been written `new IntWrapper@(PUQ)` instead. Taking the union of several partitions simply refers to a larger partition which always contains every field in each of its components.

Like Java generics, JPart types are non-variant with respect to their partition parameters. This means that two types `C@P` and `C@Q` – both of which refer to the same class, but with different partition parameters – are not subtypes of one another, even if $P \subseteq Q$. As a result, partition parameters can be referenced in both co-variant positions, such as return types, or contra-variant positions, such as the types of method parameters.

2.2 Growing Partitions

Data is added to a partition by creating new objects that contain fields located in that partition. All fields exist in exactly one partition. However, when multiple partitions are unioned together, it may not be known statically precisely which partition a field is located in. For example, if the `new` statement of Figure 1 were modified to read `new IntWrapper@(PUQ)`, then it is not defined whether the field `field` of the resulting `IntWrapper` instance is placed in `P` or `Q`; the type checker must conservatively assume that it may be in either.

2.3 Disjoint Partitions

In general, we do not require that the partition parameters be bound to disjoint partitions when a class is instantiated or a method is invoked. However, in some

cases disjoint partitions will be required by the effect checker to prove that a program is safe. Therefore, we allow the programmer to add *disjoint declarations* which name sets of partition parameters that must be mutually disjoint. An example appears in Figure 6. Note that partitions created within the current method are always known to be disjoint from all other partitions.

2.4 Partition Transfer

Creating a new object is not the only way to add data to a partition. It is also possible to change the partitioning of an existing object, so that its fields move from one partition to another. This is called *partition transfer*, and it necessitates changing the type of the object whose fields were moved to reflect the new partitioning.

Partition transfer is denoted via a cast to a type with the new partitions. The old partitions are determined from the type of the expression being cast. As an example, consider the following cast:

```
C@P variable = ...;
C@Q transferred = (C@Q) variable;
```

This has the effect of transferring any field in partition P that belongs to an object reachable from `variable` into the partition Q.

In the presence of aliasing, partition transfer is not type safe. This is because there may be extant aliases to the object whose fields were transferred, and those aliases will still have the original type. If any code should use one of those aliases, then it would assume that the object's fields were still in the original partition, when in fact they have been moved.

Rather than trying to prevent aliasing, we solve this by using the effect system to prevent the old partition from being read or written after data has been transferred out of it. Therefore, a partition transfer from P to Q means that P can never be used again. Although this restriction may seem draconian, it nonetheless enables a number of usage patterns and in particular allows data to be moved between threads.

3 Language Extensions for Parallelism

Before we proceed to discuss our effect system, it is necessary to say a few words about the threading model which we assume. As in Java, threads are created by creating a new instance of some subclass of the class `Thread`. We have chosen to use transactional memory rather than locks as the basis for synchronization. Furthermore, we have elected to replace Java's `join()` methods with a simpler, lexically scoped mechanism. To that end, we introduce two new kinds of statements:

1. The `atomic` statement, written `atomic {...}`, guarantees that all of its substatements will execute atomically, meaning without interruption by any other thread. Atomic statements are discussed in detail in [2].

```

class IntWrapper @P {
    @P int field;

    !Rd(P) int get()      { return field; }
    !Wr(P) void set(int i) { field = i;   }

    @Q !Rd(P) !Wr(Q) IntWrapper@Q clone() {
        new @R;
        IntWrapper@Q res = new IntWrapper@Q();
        res.set(field);
        return res;
    }
}

```

Fig. 2. The class `IntWrapper` which was shown before, annotated with effects.

2. The `forkjoin` statement, written `forkjoin {...}`, executes its substatements and dynamically tracks the set of threads which they start. Once all substatements have executed, the `forkjoin` statement waits for the threads which were started within to finish before it continues. It must wait not only for the threads which it has directly started, but also for any threads that they may have transitively begun themselves. `forkjoin` statements are simpler than Java's mechanism `join()` methods, but can still express real-world examples of Fork/Join parallelism [3, 4].

4 Effects

Partitions allow the program's data to be divided into distinct logical sections, but the effect system regulates how those partitions may be used.

Figure 2 shows the `IntWrapper` class defined earlier, but with each method annotated to show what effects it may have. Effect declarations precede each method declaration and are part of the method's interface. `get()` is annotated with `!Rd(P)`, as it reads a field in partition `P`, while `set()` has a corresponding write effect. Finally, `clone()` has both a `!Rd(P)` effect, as it reads `field`, and a `!Wr(Q)` effect, as it invokes `set()` on `res`, whose field is located in `Q`.

Effects are generated in two cases: dereferencing pointers and partition transfers. When dereferencing a pointer, the kind of effect depends on whether the field is read or written, and whether the modification takes place in an atomic section. The resulting four kinds of effects are `Rd(P)` and `Wr(P)`, for plain reads and writes, and `ARd(P)` and `AWr(P)`, for reads and writes which take place in an atomic statement. In each case, the parameter `P` indicates the partition that is affected, and may be either a single partition variable, or the union of several variables. Partition transfers out of a partition `P` are indicated by an effect `Xfer(P)`.

In addition to directly modifying fields, effects may be generated indirectly by invoking other methods. In this case, the effect checker uses the effects from the method's declaration to conservatively estimate the set of effects the method invocation may have. Note that if the method invocation takes place within an

atomic section, plain `Rd` and `Wr` effects are translated into their atomic equivalents.

It is the programmer's responsibility to add annotations which declare how a method may affect the in-scope partition parameters. The effect checker statically ensures that the method body cannot affect any partition parameter in a way that is not declared. It is not necessary to declare effects for partitions which are created within the method.

4.1 Effect Agreements

While effect declarations allow a programmer to limit the effects a method may have, it is sometimes useful for a method to be able to limit the effects of its caller. For example, transferring data from a partition `P` to a partition `Q` is only allowed when it can be guaranteed that `P` is not used after the transfer. If `P` is a partition parameter, however, the method needs some way to convey to its caller that it has invalidated `P` and that `P` should not be used from that point forward.

To enable these sort of guarantees, methods may be annotated with *effect agreements* that constrain what can happen after a method returns. These effects form a kind of agreement between the caller and the callee. We use a flow-sensitive effect checker to enforce them statically.

Agreements are different from Design By Contract [5]. In DBC, methods declare *preconditions*, which the caller must guarantee for the callee to function properly, and *postconditions*, which the callee promises to bring about or maintain. In this way, contract obligations flow in both directions.

In contrast, all effect agreements are obligations the callee imposes on the caller. These obligations always take the form of effects which are not permitted. Each effect agreement has a *time span* that determines precisely when the events are not permitted to occur.

In this paper, we use two different time spans for effect agreements, `post` and `par`. A `post` agreement, written `post -F(w)`¹, indicates that the effect `F(w)` may not occur at any point in the program execution after the method returns.

`post` agreements are used to ensure the safety of partition transfer. Transferring data from a partition `P` to a partition `Q` imposes a `post` agreement upon the current method, beginning at the point of transfer. The exact agreement is `post -Rd(P) -Wr(P) -ARd(P) -AWr(P) -Xfer(P)`. If the `P` partition was not created by the current method, this may require the method to declare a similar agreement so as to constrain its caller.

While sometimes necessary, `post` agreements are often stronger than is required. `par` (short for parallel) agreements can be used to limit the agreement to a shorter time span. A `par` agreement, written `par -F(w)`, indicates that a parallel thread has been started which requires that no effect `F(w)` occurs for the duration of its lifetime.

¹ The reason we use a minus sign `-` and not a `!` before the effect is to indicate that agreements describe effects which are forbidden, not permitted.

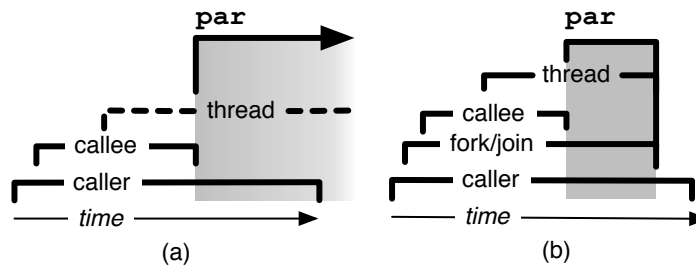


Fig. 3. A timeline demonstrating the time span to which a `par` effect agreement applies, both (a) normally and (b) in the presence of a `forkjoin` region.

Generally, we do not know when a thread will finish executing, and in these cases a `par` agreement is equivalent to a `post` agreement, as depicted graphically in Figure 3a. In the presence of a `forkjoin` statement, however, the thread's lifetime can be bounded. This is depicted in Figure 3b, where the caller knows that the thread will finish before or by the end of the `forkjoin` region.

Effect agreements are considered binding on the current thread; however, a thread is also responsible for the behavior of any thread which it (transitively) starts. Therefore, if a thread `T` invokes a method which prohibits it from later writing to a partition, then `T` may not start another thread which writes to that same partition.

4.2 Effects and Inheritance

Because effects and effect agreements form part of the interface of a method, it is important to describe how they interact with inheritance. Because it must be safe for any subtype to be used where a supertype is expected, overriding methods may not (a) have more effects than the methods they override; or (b) prohibit effects via effect agreements which are allowed in the respective supertypes. The type checker verifies these conditions statically.

4.3 Example: Multithreaded Server

One common multithreaded application is a server. Figure 4 shows a simple server which has one thread listening for connections on a given port, defined by the class `ListenerThread`. When a connection arrives, the server initializes an object describing the new connection and creates a `HandlerThread` to handle it. The handler is given control of the connection object and started in parallel. From that point on, the connection object is considered thread-local data for the handler thread, and should not be used by the listening thread anymore. In this example, we show how the effect checker, combined with effect agreements, can be used to verify that the connection object is safely transferred to the

```

class ListenerThread
  extends Thread
{
  public void run() {
    new @P;
    Socket@P socket =
      new Socket@P ();
    while (true)
      accept (socket);
  }

  !Rd(P) !Wr(P)
  @P void accept (Socket@P socket) {
    new @H;
    Socket@H connection =
      socket.accept ();
    init (connection);
    new HandlerThread@H(
      connection).start ();
  }

  !Wr(H)
  @H void init (Socket@H conn)
  { ... }
}

class HandlerThread@L
  extends Thread
{
  @L Socket@L connection;

  HandlerThread (Socket@L c) {
    connection = c;
  }

  !Rd(L) !Wr(L)
  par -Rd(L) -Wr(L)
  par -ARd(L) -AWr(L)
  public void start ()
  { ... }

  !Rd(L) !Wr(L)
  public void run ()
  { ... }
}

```

Fig. 4. The skeleton of a simple server which spawns a new thread to handle each incoming request.

new thread. In Section 6, we will expand the technique shown here into a more general solution.

The `ListenerThread` class does not have any partition parameters. Instead, within the `run()` method it creates a fresh partition, `P`, which contains the listening `Socket` instance. The `HandlerThread` class is parameterized by a single partition `L` for its thread-local data. In its `accept()` method, the `ListenerThread` creates a new partition `H` which it gives to each new `HandlerThread` to use for its thread-local data.

The error we are trying to prevent is that the `ListenerThread` continues to write to the partition `H` after it has started the `HandlerThread`. To prevent this, the `HandlerThread` has declared effect agreements on its `start()` method which prohibits its local partition `L` (`H`, from the `ListenerThread`'s point of view) from being read or written. These agreements are given `par` scope so that they are in effect as long as the thread may execute.²

To verify that effect agreements are respected, the effect checker computes what effects may occur after each statement in the method. The result of this computation for the `accept()` method of `ListenerThread` is shown in Figure 5.

Since we are computing what events are to come, the analysis is done starting at the end of the method and working backwards. Therefore, line 12 depicts

² A sharp-eyed reader will note that, because the `start()` method is inherited from `Thread`, `HandlerThread` cannot add effect agreements in this fashion. We resolve this in Section 6 by modifying the thread class itself.

```

1  !Rd(L) !Wr(P)
2  @P void accept(Socket@P socket) {
3      // { Rd/Wr(P) }
4      new @H;
5      // { Rd/Wr(P) Rd/Wr(H) }
6      Socket@H connection = socket.accept@H();
7      // { Rd/Wr(P) Rd/Wr(H) }
8      init@H(connection);
9      // { Rd/Wr(P) Rd/Wr(H) }
10     new HandlerThread@H(connection).start();
11     // { Rd/Wr(P) }
12 }

```

Fig. 5. The results of a flow-sensitive effect analysis of the `accept()` method from Figure 4.

the initial set of effects. Because there are no effect agreements declared on this method concerning `P`, we must make the conservative assumption (which happens to be true, in this case) that data in partition `P` may be both read and written in the future. Note that there no conservative assumptions are needed for `H` as it is newly created in this method.

Line 10 contains the call which starts the `HandlerThread`. This is where we must verify the effect agreements for `start()`: to do so, we compare the set of events to come from line 11 with the forbidden events, and determine that the method call is permitted. Since `start()` declares that it reads and writes `HandlerThread`'s partition parameter, we determine that `H` may be both read and written at this point and add those effects to the set, yielding line 9.

From line 9 back to line 5, the effect set is unchanged because it already contains reads and writes. No methods are invoked which declare an effect agreement, so there is no need to check for conflicts. Finally, we reach line 4 which creates the `H` partition and therefore remove the `Rd/Wr(H)` effects, leaving only effects on `P` in line 3.

5 Effect Checker

This section describes the design of the effect checker, the module responsible for verifying that methods properly declare any effects they may have and that they abide by any effect agreements due to method calls or partition transfers.

5.1 Checking the Method Interface

Verifying that the method implementation obeys the bounds of its interface is done with a standard, flow-insensitive analysis. We simply take the union of all effects generated by any statement in the method body, and verify that, for each effect $F(P)$, either (a) `P` is a created by a `new @P` statement in this method, or (b) the method declares an effect $F(Q)$, where $P \subseteq Q$.

```

OUT  $\leftarrow$  {  $\emptyset$  for each block }
initialize OUT with starting assumptions
while OUT continues to change do
  for all blocks  $b$  do
     $f_{\text{succ}} \leftarrow \cup(\text{OUT}_{b'} \text{ for each } b' \in \text{successors of } b)$ 
     $f_{\text{stmts}} \leftarrow$  the set of effects caused by any statement in  $b$ 
    if checking a par agreement and  $b$  terminates a basic block then
       $f_{\text{stms}} \leftarrow \emptyset$ 
    end if
     $\text{OUT}_b \leftarrow f_{\text{succ}} \cup f_{\text{stms}}$ 
  end for
end while

```

5.2 Flow-Sensitive Effect Analysis

Because effect agreements constrain the events which can occur within a specific period of time, a flow-sensitive analysis is required to check them. For this purpose, we use an iterated analysis that propagates sets of effects around the control flow graph until it reaches a fixed point. Similar algorithms are commonly used for data-flow analysis in compilers [6]. Because effect agreements constrain what happens after a method returns, we use a reverse analysis, with set union as the confluence operator for joining multiple control flows.

The effect flow algorithm is shown as Algorithm 5.2. This algorithm is in fact executed twice, once for **par** effect agreements and once for **post** agreements. For each basic block b , it computes a set OUT_b which contains the set of effects that could occur in the current time span. When checking **post** agreements, therefore, OUT_b contains the set of effects that can occur once b begins execution. When checking **par** agreements, on the other hand, it contains the set of effects that might occur in parallel with a thread that starts right before the control flow reaches b .

Computing the OUT set for a basic block b begins by taking the union of the OUT sets for each successor of b (except for a slight twist regarding **forkjoin** statements, discussed in detail below). The effects which may occur due to statements within b are then added to this set, resulting in the new value for OUT_b . This process repeats for all basic blocks until it reaches a fixed point.

5.3 Starting Conditions

To start the algorithm, the initial set of effects at the end of the method are defined conservatively. We assume that any effect may occur which is not specifically forbidden by an effect agreement of the appropriate time span. Therefore, when performing **post** analysis, the starting set contains all possible effects except those barred by some **post** agreement. Likewise, during the **par** analysis, only those effects barred by **par** agreements are removed from the starting set.

```

1  abstract class Thread@S@R@L disjoint(S,R,L) {
2
3      !ARd(S ∪ R ∪ L)
4      !AWr(S ∪ L)
5      !Rd(R ∪ L)
6      !Wr(L)
7      abstract void run();
8
9      !ARd(S ∪ R ∪ L)
10     !AWr(S ∪ L)
11     !Rd(R ∪ L)
12     !Wr(L)
13     par -Wr(S ∪ R ∪ L)
14     par -Rd(S ∪ L)
15     par -AWr(R ∪ L)
16     par -ARd(L)
17     par -Xfer(S ∪ R ∪ L)
18     abstract void start();
19
20 }

```

Fig. 6. The definition of legal effects for a **Thread**. Using these effects guarantees that all partitions modified by a thread are used in a safe fashion.

5.4 forkjoin statements

When checking **par** agreements, the **forkjoin** statement is given a special significance. This is because it is statically known that all threads beginning within a **forkjoin** statement will have terminated by the time the **forkjoin** statement finishes. This means that, when checking **par** agreements, the basic block which terminates the **forkjoin** statement may safely disregard the effects of its successors, because it is known that they will not occur in parallel with any threads that start within.

6 Enforcing Thread-Safety

We can use effect agreements to ensure that a partition is never accessed by multiple threads in an incompatible fashion. The core idea is to use the type of the **Thread** object, from which all threads must derive, to guarantee that threads only use partitions in one of several pre-approved ways.

We guarantee that all partitions the thread may access fall into one of three categories:

1. *Thread-Local* partitions are read and written by the thread outside of **atomic** sections. Such partitions may not be simultaneously accessed by other threads.
2. *Read-Only* partitions are never written by the thread and are read outside of **atomic** sections. Such partitions may not be written by other threads.
3. *Shared* partitions are modified by multiple threads simultaneously. Accesses to shared partitions, read or write, must take place within an **atomic** section.

We assume that threads are started by invoking the method **start()** defined in the **Thread** class. The actions of a thread are defined by its **run()** method;

invoking `run()` directly, however, does not start the thread, but is merely a normal method call that runs in the current thread.

Therefore, we can use the declared interface on the `start()` and `run()` methods to control what threads are permitted to do. The desired interface for class `Thread` is shown in Figure 6. The idea is to parameterize the thread by three partitions, `S`, `R`, and `L`, which contain respectively the shared, read-only, and thread-local data that this thread may access. As we will see, the definition of the `run()` and `start()` methods do not permit data in these partitions to be used in any way other than those outlined above. Furthermore, because a method must list all of its effects in its interface, we can be sure that this thread will not affect any partitions other than `S`, `R`, or `L`.

To get a better understanding of the definition in Figure 6, let us examine it piece by piece. The first line declares the partition parameters, `S`, `R`, and `L`, and states that they must be mutually disjoint.

The `run()` method is defined on lines 3–7. The effect declarations which precede it describe how the thread is allowed to access `S`, `R`, and `L`. Line 3 indicates that any partition may be atomically read, whereas line 4 restricts atomic writes to shared (`S`) and thread-local (`L`) data. Line 5 allows read-only (`R`) and thread-local (`L`) data to be read non-atomically, but only `L` may be modified in a non-atomic fashion, as indicated on line 6.

The `start()` method is defined next. Lines 9–12 indicate that the `start()` method has the same effects as the `run()` method. Lines 13–17 define the effect agreement for the `start()` method. These agreements highlight the important difference between `start()` and `run()`: invoking `run()` does not actually start a second thread. `start()`, however, performs its actions in parallel with the current thread, and therefore it has to place constraints on the current thread. Note the use of the `par` time span for these effect agreements, which guarantees that the forbidden events will not occur in parallel with this thread, though they may occur after the thread is known to have terminated.

Line 13 guarantees that the parent thread does not write non-atomically to any partition that the child thread has access to. Line 14 guarantees that the parent thread does not try to read non-atomically from any partition which the child thread will be writing to. Line 15 guarantees that the parent thread will not make atomic writes to the `R` or `L` partitions. Line 16 guarantees that the thread-local data is not atomically read by the parent thread. Finally, line 17 guarantees that no data is transferred out of the shared, read-only, or local partitions while the thread is active.

At first, it might seem stringent to require that every `Thread` class describe their data in exactly three partitions. However, due to the possibility of using partition expressions as parameters, this is not a real limitation. For example, to define a thread which has a shared partition `S`, no read-only partition, and two local partitions, `L1` and `L2`, one can extend `Thread@S@/(L1UL2)`.

```

class Main {
    @A !Rd(A)
    void mapreduce(List@A list) {
        new @O;
        List@O results = new List@O ();
        forkjoin {
            for (Object@A obj : list) {
                new @L;
                MapThread@O@A@L mt =
                    new MapThread@O@A@L ();
                mt.inObject = obj;
                mt.results = results;
                mt.start ();
            }
        } // reduce results,
        // without atomic
    }
}

class MapThread@S@R@L
    extends Thread@S@R@L
{
    @L Object@R inObject;
    @L List@S results;

    !Rd(R@L) !Wr(L) !ARd(S) !AWr(S)
    void run() {
        new @T;
        // Map in Object to tmp:
        Object@T tmp = ...;
        // Partition transfer:
        Object@S tmp2 = (Object@S) tmp;
        // Add to output array:
        atomic { results.add(tmp2); }
    }
}

```

Fig. 7. Map-reduce example

6.1 Example: Map Reduce

Figure 7 shows a more involved example following the well-known map reduce pattern [7]. The map reduce pattern uses multiple threads to perform some mapping operation on each item in an array in parallel. After all the threads have completed, it then performs a reduce action which combines the output of the map stage.

The example contains two classes `Main` and `MapThread`. The `Main` class is given a list of objects `list`, contained in some partition `A`, and creates a `MapThread` instance for each item contained within. All of the threads share access to the `O` partition, which is where they will output the mapped results. They also share read-only access to the `A` partition containing the input array and its items. Finally, a new partition `L` is created for each thread and used to store its local data.

The `MapThread`'s `run()` method begins by creating a temporary partition, `T`. It then performs its mapping operation, creating an output value `tmp` located in the partition `T`. In order to give the main thread access to `tmp`, it transfers the object into the shared partition and adds it to the output list `results`. The actual modification to the shared list must take place within an atomic section.

Note that, once the `MapThreads` have completed, the `Main` class is able to access the `results` array and the objects within without synchronization. This is due to the `forkjoin` region, which ensures that all parallel threads have finished, and therefore limits the scope of the effect agreements by which `Main` is bound.

7 Related Work

The work in this paper is in many ways a synthesis of several existing techniques, and therefore touches on many different bodies of work. As we do not have the space to do justice to all of it, we focus here on that which is most closely related.

Guava [8] is a dialect of Java that does not permit data races by construction. Guava’s constructs and type system embody several safe, best practices for multi-threaded program. In this sense, it is similar to the restrictions we placed on the `Thread` class, which require that each partition be categorized into one of several pre-approved patterns.

RccJava [9] is a static type checker for Java programs which is able to detect race conditions. Its type system is based on locks, and the tool has been used to verify an impressive body of existing code. Our flow-sensitive effect analysis is able to capture patterns that they cannot, however. For example, we allow all objects to start as thread-local when first created, but transition to a shared state when they become reachable from another thread.

SafeJava [10] enforces the consistent use of a deadlock-free locking discipline through their ownership type system. The locking discipline is based on the ownership structure, and requires that an object’s owner be locked before the object can be accessed. This approach entangles encapsulation and threading, which may require that one or the other be compromised. Fine-grained locking schemes, for example, require a flat ownership structure, which then provides weaker encapsulation guarantees. In contrast, our approach strives to separate the partitioning structure of the program from the thread safety check.

Effect systems have a long history in the literature, beginning with the work of Lucassen and Gifford [11]. Our treatment of effects has much in common with other object-effect systems [12, 13] and in particular with the flow-sensitive effect system described in Contextual Effect Systems [14]. One important distinction of our work from prior work is that we include a mechanism for the programmer to describe not only the effects of a method, but also the potential causes of interference between methods.

Another approach to controlling effects is through the use of capabilities, which limit how a particular reference can be used [15–17]. For example, such capabilities might prevent writes or enforce uniqueness.

Our approach to partition transfer is similar in spirit to [18], which enforces uniqueness not by forbidding aliases, but by requiring that all aliases are dead at the point where an object must be unique.

8 Conclusion

In this paper, we have presented *partitions*, an abstraction for exposing the alias structure of a program, along with an accompanying flow-sensitive effect system with effect agreements. We also detail how our effect system can be used to check that a multi-threaded program uses safe patterns for its synchronization.

Our long-term goal is to give programmers a simple and expressive way to check semantic properties of their own design. Rather than encoding a specific notion of correctness into the type system, we aim to develop generic mechanisms, such as partitions and effect agreements, that can be reused for a variety of purposes.

References

1. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News* **21** (1993) 289–300
2. Harris, T., Fraser, K.: Language support for lightweight transactions. *SIGPLAN Not.* **38** (2003) 388–402
3. Bull, J.M., Kambites, M.E.: JOMP—an OpenMP-like interface for Java. In: *JAVA*, ACM (2000) 44–53
4. Lea, D.: A Java fork/join framework. In: *JAVA*, ACM (2000) 36–43
5. Meyer, B.: *Object-Oriented Software Construction*. Prentice Hall PTR (2000)
6. Aho, A.V., Ullman, J.D.: *Principles of Compiler Design*. Addison-Wesley Longman Publishing Co., Inc. (1977)
7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: *OSDI, USENIX Association* (2004)
8. Bacon, D.F., Strom, R.E., Tarafdar, A.: Guava: a dialect of Java without data races. In: *OOPSLA*, ACM (2000) 382–400
9. Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.* **28** (2006) 207–255
10. Boyapati, C.: Safejava: a unified type system for safe programming. PhD thesis, Massachusetts Institute of Technology (2004) Supervisor-Martin C. Rinard.
11. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: *POPL '88*, ACM (1988) 47–57
12. Greenhouse, A., Boyland, J.: An Object-Oriented Effects System. In: *ECOOP*, Springer-Verlag (1999) 205–229
13. Rustan, K., Leino, M.: Data groups: specifying the modification of extended state. In: *OOPSLA*, ACM (1998) 144–153
14. Neamtiu, I., Hicks, M., Foster, J.S., raticakis, P.P.: Contextual effects for version-consistent dynamic software updating and safe concurrent programming. *SIGPLAN Not.* **43** (2008) 37–49
15. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: *OOPSLA*, ACM (2002) 311–330
16. Boyland, J., Noble, J., Retert, W.: Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In: *ECOOP*, Springer-Verlag (2001) 2–27
17. Noble, J., Vitek, J., Potter, J.: Flexible Alias Protection. In: *ECOOP*, Springer-Verlag (1998) 158–185
18. Boyland, J.: Alias burying: unique variables without destructive reads. *Softw. Pract. Exper.* **31** (2001) 533–553