Static Analysis of Dynamic Schedules and its Application to Optimization of Parallel Programs

Christoph M. Angerer and Thomas R. Gross

ETH Zurich, Switzerland

Abstract. Effective optimizations for concurrent programs require the compiler to have detailed knowledge about the scheduling of parallel tasks at runtime. Currently, optimizations for parallel programs must define their own models and analyses of the parallel constructs used in the source programs. This makes developing new optimizations more difficult and complicates their integration into a single optimizing compiler.

We investigate an approach that separates the static analysis of the dynamic runtime schedule from subsequent optimizations. We present three optimizations that are based on the information gathered during the schedule analysis. Variants of those optimizations have been described in the literature before but each work is built upon its own highly specialized analysis. In contrast, our independent schedule analysis shows synergistic effects where previously incompatible optimizations can now share parts of their implementation and all be applied to the same program.

1 Introduction

With the arrival of multicore systems, parallel programming is becoming increasingly mainstream. Despite this, compilers still remain largely ignorant of the task scheduling at run-time. Absent this knowledge, however, a compiler is missing important optimization and verification opportunities.

Because compilers do not have a good understanding of the runtime scheduling of tasks, researchers developing optimizations for parallel programs must additionally develop their own model and analysis of the parallel constructs in use. The overhead of defining a full-fledged analysis, however, obfuscates the actual optimization and prohibits synergistic effects that might emerge by combining different optimizations.

In this paper, we present a static analysis for parallel programs that is independent from any concrete optimization. By using our schedule analysis, the core algorithms of existing optimizations can often be implemented in only a few simple rules. A small algorithmic core not only helps with a better understanding of the optimization but also supports their integration into a single optimizing compiler. The contributions of this paper are:

- we describe a model for fine-grained parallelism based on lightweight tasks with explicit scheduling. This model is powerful enough to express a wide variety of existing parallelism constructs (Section 2);
- we develop a schedule analysis that computes an abstract schedule from a program containing explicit scheduling instructions (Section 3);
- we present three optimizations for parallel programs that make use of the results of the schedule analysis. Two optimizations work with fundamentally different synchronization primitives: locks and transactional memory. The common schedule analysis enables the optimization of programs that intermix both those synchronization paradigms (Section 4).

The core of the schedule analysis has been implemented in a prototype and is available online at *http://wiki.github.com/chmaruni/xsched/* (Section 5).

2 A Program Model with Explicit Scheduling

In this section we describe a model for fine-grained parallelism based on lightweight tasks with explicit scheduling. This model is general enough to express a wide variety of existing concurrency patterns, from structured fork-join style parallelism to unstructured threads. The explicit happens-before relationships simplify the analysis of parallel program schedules while avoiding the limitations of lexically scoped parallelism.

The basic building block of our execution model is a *task*. A task is similar to a method in that it contains code that is executed in the context of a `this`-object (or the class, in the case of `static` methods/tasks). Unlike a method, however, one does not *call* a task, which would result in the immediate execution of the body, but instead *schedules* it for later execution.

As an example, consider a task `t()` that starts a long-running computation `compute()` and schedules a task `print()` that will print the result after the computation has finished:

```
task t() {
    Activation aCompute = schedule(this.compute());
    Activation aPrint = schedule(this.print());
    aCompute→aPrint;
}
```

A schedule is represented as a graph of $\langle object, task() \rangle$ pairs. The statement `schedule(this.print())`, e.g., creates a new node with the `this` object and the `print()` task and returns an object of type `Activation` representing that node. Like any other object, `Activation` objects can be kept in local variables, passed around as parameters, and stored in fields.

At runtime, a scheduler constantly chooses activations that are eligible for execution and starts them. The order in which the scheduler is allowed to start the activations is specified by the edges in the schedule graph. If the schedule contains a happens-before edge $\langle o1, t1() \rangle \rightarrow \langle o2, t2() \rangle$, the scheduler must guarantee that activation $\langle o1, t1() \rangle$ has finished execution before activation $\langle o2, t2() \rangle$

```
1   class ParallelOrderedMap {
2       Vector out;
3
4       private task doMap(Object data) {
5           Object mappedData = //complex computation using data
6           now.result = mappedData;
7       }
8       private task doWrite(Activation mapActivation) {
9           out.add(mapActivation.result);
10      }
11      task mapInput(Vector input) {
12          Activation lastWrite = now;
13          for(Object data : input) {
14              Activation map = schedule(this.doMap(data));
15              Activation write = schedule(this.doWrite(map));
16
17              map → write;
18              lastWrite → write;
19
20              lastWrite = write;
21      } } }
```

Fig. 1. Example of a parallel ordered mapping operation.

is started. The statement aCompute→aPrint creates an explicit happens-before relationship between the two activation objects aCompute and aPrint.

In a program, the currently executing activation can be accessed through the keyword now. Whenever a new task is scheduled, the scheduler automatically adds an initial happens-before relationship between now and the new activation node. This initial edge prevent the immediate execution of the new activation and allows the current task to add additional constraints to the schedule before it finishes. Therefore, in the example the scheduler implicitly creates two additional edges now→aCompute and now→aPrint.

2.1 Example of a Parallel Ordered Mapping Operation

Figure 1 shows a more complex example of a class implementing a parallel ordered mapping operation. When the mapInput() task is activated, passing a Vector of input elements, this class will apply a (possibly expensive) mapping operation to each input element in parallel and write the resulting mapped values into the out vector in the original order.

The loop on line 13 iterates through every element in the input vector and, for each element, schedules the doMap() task on line 14, passing the data element to the activation. Line 15 then activates the doWrite() task for every element. A chain of doWrite() activations write the mapped values in the correct order into the out vector.
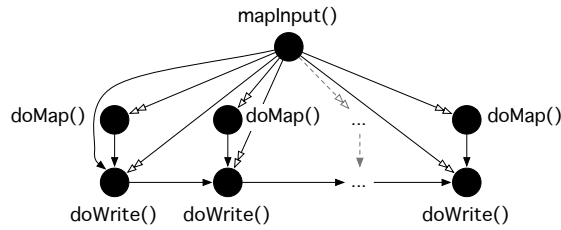
Fig. 2. The schedule created by `mapInput()` from Figure 1.

For `doWrite()` to get to the result of `doMap()`, we pass the `map` activation object as a parameter to `doWrite()`. Inside `doMap()` we use the `result` field provided by `Activation` to store the result of the mapping operation which can then be read in `doWrite()`. The explicit happens-before relationship added on line 17 ensures that the mapped value has been stored in the `result` field before `doWrite()` executes.

So far we ensured the ordering between the mapping operation and the write operation for each single element. The correct ordering of the writes is achieved by an additional happens-before relationship between the current `write` activation and the `lastWrite` activation on line 18. Initially, we set `lastWrite` to `now` in line 12 and then update it to the most recent `write` activation on line 20.

Figure 2 shows the schedule that is created by `mapInput()`. The double-headed arrows are created implicitly by the `schedule` statements whereas the single-headed arrows stem from the →-statements.

After `mapInput()` has finished setting up the schedule, the scheduler can choose any of the `doMap()` activations for parallel execution because they are not ordered with respect to one another. The `doWrite()` activations must be executed in order, however, which guarantees the correct ordering of the written values in the `out` vector.

2.2 Additional Synchronization Primitives

Explicit scheduling alone is expressive enough to model many concurrency patterns such as fork-join, bounded-buffer producer-consumer, or fuzzy barriers [9].

There are cases, however, that require nondeterministic choice—which cannot be expressed with explicit scheduling alone. Non-deterministic access to shared resources requires additional synchronization primitives such as atomic compare-and-swap operations, locks, or transactional memory. Take, for example, a shared resource such as a printer and two parallel tasks waiting for user input before accessing the printer. There is no point in the program where we can define an ordering between the two tasks beforehand, because the timing of the user input is unknown.
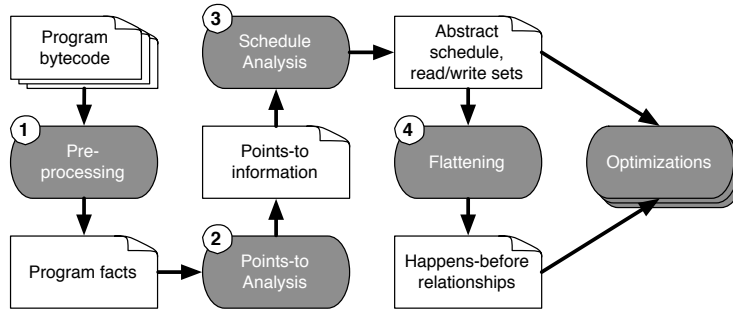
Fig. 3. The phases of a schedule analysis.

3 Schedule Analysis

The core of our approach is a schedule analysis that can determine whether two activations are executed in parallel, sequentially, or exclusively. Schedule analysis thus computes the function $Activation \times Activation \rightarrow Relation$ where $Relation$ is one of the following:

Sequential: Two activations are sequential if their execution is strictly ordered.

Exclusive: Two activations are exclusive if they can never co-exist in a single run of the program (e.g., they are scheduled in different branches of a conditional statement).

Parallel: If two activations are neither sequential nor exclusive, they are considered (potentially) parallel.

In addition, schedule analysis computes the sets of objects that are read and/or written by each activation. The information about the read- and write-sets together with the information about the relative ordering of activations can then be used by subsequent optimizations such as the ones from Section 4.

The algorithm presented in this paper computes the abstract schedule and the read-/write-sets in the four phases shown in Figure 3. Given the bytecode of a whole program as input, a pre-processing phase extracts static information about the program structure. This information is used by a standard points-to analysis to propagate alias information and compute points-to sets for object fields and program variables. Points-to information is necessary because activations are normal objects that can be stored in fields and passed as parameters.

The third phase is the schedule analysis. During this phase we compute the read- and write-sets for each activation and extract an abstract schedule from the unconditional →-statements present in the program. The abstract schedule is a directed graph with different node and edge types. The fourth phase takes this graph and flattens it into the binary relations for sequential, exclusive, and parallel activations.

V	the domain of variables. V contains all the allocation sites, formal parameters, return values, thrown exceptions, cast operations, and dereferences in the program.
H	the domain of heap objects. Heap objects are named by the invocation sites of object creation operations.
F	the domain of fields in the program. There is a special field *elements* to denote an array access.
M	the domain of implemented methods in the program. It does not include abstract or interface methods.
N	the domain of virtual method names used in invocations.
BC	the domain of bytecodes in the program.

Fig. 4. The Datalog domains.

3.1 Datalog

Most phases of our analysis are formulated and implemented as Datalog programs. We chose Datalog because it is a concise high-level specification language that has been shown to be well-suited for dataflow analyses and scalable to even large real-world programs [18].

The basis of Datalog are two-dimensional tables called *relations*. In a relation, the columns are the attributes, each of which is associated with a finite domain defining the set of possible values, and the rows are the tuples that are part of this relation. Figure 4 shows the domains that we use in this paper.

If tuple (x, y, z) is in relation A, we say that predicate $A(x, y, z)$ is true. A Datalog program consists of a set of rules that compute new members of relations if the rule body is true. E.g., the rule:

```
D(w, z) : A(w, x), B(x, y), !C(y, z).
```

says that "tuple (w, z) is added to D if $A(w, x)$, $B(x, y)$, and not $C(y, z)$ are all true." The Datalog runtime will apply rules until a fixed point has been reached and no more tuples can be added to the relations.

3.2 Pre-processing and Points-to Analysis

The first two phases of the analysis implement a standard points-to analysis. For space reasons, we only describe briefly the outcomes of both phases. An in-depth explanation of the algorithms we use is given in [18].

Before the points-to analysis (implemented as a Datalog program) starts, a pre-processor extracts information about the analyzed program and generates input tuples that can be read by Datalog. The pre-processor generates many different relations encoding information about type hierarchies, virtual method calls, object creation, and more; in this paper, however, we are mostly interested in the following two relations:

store : $BC \times V \times F \times V$ represents store statements. $store(bc, v1, f, v2)$ says that bytecode bc is a statement "v1.f = v2".

load : $BC \times V \times F \times V$ represents load statements. $load(bc, v1, f, v2)$ says that bytecode bc is a statement "v2 = v1.f".

The goal of the points-to analysis is to compute the following three relations from the input relations generated by the pre-processor:

variablePT : $V \times H$ is the variable points-to relation. $variablePT(v, obj)$ means that variable v can point to heap object obj.

heapPT : $H \times F \times H$ is the heap points-to relation. $heapPT(obj1, f, obj2)$ means that field f of heap object $obj1$ may point to heap object $obj2$.

invocationEdge : $BC \times M$ is the relation of the resolved targets of invocation sites. $invocationEdge(bc, m)$ says that invocation bytecode bc may invoke the method implementation m.

During the points-to analysis we treat `schedule`-statements as normal method calls. This works because all the parameters for the activation are bound when the `schedule` statement is executed even though the exact time when the activation will execute is not known. $\rightarrow$-statements are ignored during this phase.

Whaley and Lam [18] describe various points-to analyses with a variety of trade-offs between precision and computational cost. For the rest of the paper we assume a context insensitive analysis with on-the-fly call graph discovery, but other, more precise variants can be used.

3.3 Computing and Flattening the Abstract Schedule

The main task of the schedule analysis is to compute an abstraction of the scheduling graphs that can occur at runtime. For this, the analysis must take the `schedule`-statements for the initial creation edges and all $\rightarrow$-statements for additional happens-before edges into account.

In general, the safe and conservative assumption is to over-approximate parallelism. As an example, take the detection of data races. Two activations are allowed to write to the same data if and only if they are sequentially ordered. If the sequential execution cannot be guaranteed we must assume that both tasks are potentially executed in parallel and report a data race if they access the same data.

Because the analysis cannot rely on happens-before relationships that are created conditionally, we only consider unconditional $\rightarrow$-statements. Further, if for a statement `lhs`$\rightarrow$`rhs` the points-to analysis found that one or both variables `lhs` and `rhs` may point to more than one activation object, the abstract schedule must over-approximate parallelism by ignoring the happens-before edge because it cannot guarantee the exact ordering of the involved activations.[1]

The core of the abstract schedule computation can be expressed in the following Datalog rules:

---

[1] In this case, increasing the context-sensitivity can result in higher precision and therefore smaller points-to sets which may allow the analysis to drop less edges.

```
multiple(v)   :- variablePT(v, obj1), variablePT(v, obj2), obj1 != obj2.
singleton(v) :- !multiple(v).

happensBeforeEdge(source, target) :-
                arrowStatement(lhs, rhs),
                variablePT(lhs, source), variablePT(rhs, target),
                singleton(lhs), singleton(rhs).
```

The relation $multiple : V$ contains all the variables that may point to two (or more) different objects whereas the relation $singleton : V$ contains all variables not in $multiple$ and thus pointing to at most one object. Given an arrow-statement lhs→rhs, the $happensBeforeEdge : H \times H$ relation contains the tuple (source, target) if the variables lhs and rhs point to the singleton objects source and target respectively.

The points-to analysis can only track a finite number of heap objects (including activation objects) but a program that contains loops and recursion can create a potentially infinite number of objects. For this reason, filtering out ambiguous →-statements is a necessary but not sufficient condition for computing a conservative abstract schedule because a single object at analysis time may represent multiple runtime objects.

In the example from Figure 1, there are potentially many activation objects created at lines 14 and 15. Therefore, the happens-before edge map→write on in line 17 is only valid without restrictions inside the same loop iteration. An activation doMap() of a later iteration, e.g., is not guaranteed to happen before a doWrite() activation of an earlier iteration.

To address this problem, we make use of the fact that a program in static single assignment form (SSA) captures the flow of values between loop iterations in the form of explicit Φ operations. The pre-processor described in Section 3.2 treats a statement var3 = Φ(var1, var2) in the source program similar to an object creation site. That is, it adds a new object phiObj to H and records the assignment of phiObj to var3. Additionally, the preprocessor adds the facts varIntoPhi(var1, phiObj) and varIntoPhi(var2, phiObj) to a relation $varIntoPhi : V \times H$ indicating that variables var1 and var2 flow into the phiObj.

With this information in place, the schedule analysis can compute the relation $phiEdge : H \times H$ with the following rule:

```
phiEdge(actObj, phiAct) :-
                varIntoPhi(actVar, phiAct), variablePT(actVar, actObj).
```

A fact phiEdge(act, phi) means that the activation heap object act flows into the phi heap object phi.

Figure 5 shows the mapInput()-task from Figure 1 and the abstract schedule that is computed by the schedule analysis. Solid nodes represent normal activation nodes and dashed nodes are Φ activation objects. The dashed edges are computed by the above phiEdge rule. In addition to the activation nodes, the graph contains dashed boxes indicating loop boundaries: a solid node is inside a
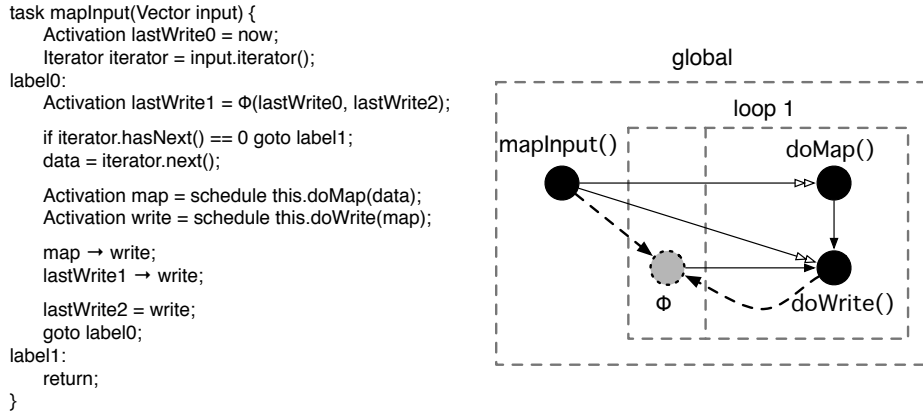
```
task mapInput(Vector input) {
    Activation lastWrite0 = now;
    Iterator iterator = input.iterator();
label0:
    Activation lastWrite1 = Φ(lastWrite0, lastWrite2);

    if iterator.hasNext() == 0 goto label1;
    data = iterator.next();

    Activation map = schedule this.doMap(data);
    Activation write = schedule this.doWrite(map);

    map → write;
    lastWrite1 → write;

    lastWrite2 = write;
    goto label0;
label1:
    return;
}
```


Fig. 5. The `mapInput()` task from Figure 1 in SSA form and the abstract schedule.

dashed box if the `schedule`-statement is inside the loop body. The Φ nodes that play the role of loop variables are placed in a special "header" area in the loop. The loop information is computed by a *structural analysis* on a global interprocedural SSA graph [10]. A special `global` box represents the whole program.

With the graph in Figure 5, we can deduce that the `mapInput()` activation always happens before both the `doMap()` and the `doWrite()` activations. This is because `mapInput()` is in the `global` context and connected to the other two activations by creation edges. Further, the `doWrite()` activation is sequential to itself because of the recursion through the Φ node. The recursion loop encodes the fact that all `doWrite()` activations are ordered with respect to one another.

The `doMap()` activation, on the other hand, is parallel to itself, because it is created inside a loop, as well as parallel to `doWrite()` because the happens-before edge between them is created inside the `loop 1` box and therefore has no effect on the `global` context.

This example demonstrates that the effect of a happens-before edge generally depends on the loop context: in `loop 1`, we can say that `doMap()` always happens-before `doWrite()` because those objects are created inside this loop; but this is not true for the `global` perspective. Only Φ nodes allow us to establish happens-before relationships across loop iterations.

Most optimizations work in the `global` context, because they transform the source code which affects the whole program and not only a single loop iteration. Therefore, the last phase of the schedule analysis flattens the abstract schedule in the `global` context. The flattening process creates the three relations *parallel* : $H \times H$, *sequential* : $H \times H$, and *exclusive* : $H \times H$ by using the abstract schedule to find the type of relationship for each pair of activation objects.[2]

---

[2] Deciding exclusivity requires further flow-sensitive analysis of the source code but it can reduce the number of activations that are unnecessarily classified as being potentially parallel.

3.4 Computing Read- and Write-Sets

The second part of the schedule analysis is to compute the read- and write-sets for each activation. We capture the read and write sets in relations $read : H \times BC \times H$ and $write : H \times BC \times H$. A tuple `read(act, bc, obj)`, e.g., states that activation object `act` may reach bytecode `bc` and this bytecode is a `load` that may access object `obj`. The computation of the read and write sets is straightforward and can be expressed in the following two Datalog rules (where an underscore '_' means "any"):

```
read(act, bc, obj) :-
      activationReaches(act, bc), load(bc, v, _, _),  variablePT(v, obj).
write(act, bc, obj) :-
      activationReaches(act, bc), store(bc, v, _, _), variablePT(v, obj).
```

The relation $activationReaches : H \times BC$ is a simple reachability predicate that starting from the `task` of an activation object follows all invocation edges in the call graph to find all bytecodes that this activation may execute.

4 Optimizations Based on Schedule Analysis

In this section, we present three sample optimizations for parallel programs that are all based on the same schedule analysis from Section 3. The first two optimizations have been taken from the literature and target the two main synchronization primitives, locks and transactional memory. The third optimization is specific to our explicit scheduling model and tries to reduce the number of happens-before relationships in a program thus reducing scheduling overhead and potentially increasing parallelism.

4.1 Synchronization Removal

Like many imperative and object-oriented languages, Java provides a synchronization mechanism based on locks. Whenever a method or block may access data structures that are shared between multiple threads, the programmer must guard the critical section with a lock, e.g., using the `synchronized` keyword. Because a thread-safe library cannot know the context it is used in, it must conservatively assume a multi-threaded environment and guard all critical sections that potentially access shared data. In many programs, however, a large number of the locking operations may safely be removed because two parallel tasks never contend for the same locks.

A critical section is required if two parallel activations `act1` and `act2` may try to acquire a lock on the same object `lockObj`. Acquiring a lock requires the execution of a dedicated monitor enter instruction that is associated with a variable pointing to the lock object. Conversely, a critical section is unnecessary if its guarding monitor enter is not required. The following Datalog rules compute the set of required and unnecessary monitor enter bytecodes:

```
lockObject(monitorEnterBC, obj) :-
                lockVariable(monitorEnterBC, v), variablePT(v, obj).
requiredMonitorEnter(monitorEnterBC1) :-
                parallel(act1, act2),
                activationReaches(act1, monitorEnterBC1),
                activationReaches(act2, monitorEnterBC2),
                lockObject(monitorEnterBC1, lockObj),
                lockObject(monitorEnterBC2, lockObj).
unnecessaryMonitorEnter(monitorEnterBC) :-
                !requiredMonitorEnter(monitorEnterBC).
```

If, in the example from Figure 1, the programmer had guarded the call `out.add()` in the `doWrite()` task with a lock, the analysis would consider this lock as unnecessary because all activations of `doWrite()` are ordered and the `parallel(act1, act2)` clause is always false for two `doWrite()` activations. If the programmer had also guarded the body of task `doMap()` with the same lock, the above rules would consider all locks to be required because `doMap()` activations can happen in parallel with other `doMap()` and `doWrite()` activations.

Ruf [13] describes the same optimization but based on an analysis algorithm that is specialized to the task of synchronization removal. One of the achievements is that this approach can remove 100% of all synchronization for the special case of single threaded programs. Looking at the rules above, we can see that our optimization has the same property. In a single threaded program, the clause `parallel(act1, act2)` is always false and therefore all monitor enter bytecodes will be classified as unnecessary.

4.2 Reducing Strong Atomicity Overhead

Transactional memory is a promising alternative to synchronization that avoids many of the problems associated with locks. In a TM system, an atomic region `atomic{b}`, where the block `b` is a list of statements, requires the runtime to execute the sequence `b` *as though* there were no interleaved computation. When the transaction inside the atomic region completes, it either *commits*, thus making the changes visible to other processes, or it *aborts*, causing the transaction to be rolled back and the atomic region to be re-executed.

A transactional system is said to have *weak* atomicity semantics if it allows computations outside of transactions to be interleaved with transactions. Weak semantics allow for a more efficient implementation but it sacrifices ordering and isolation guarantees which can lead to incorrect execution of programs that are correctly synchronized under locks [16].

Strong atomicity, on the other hand, requires memory accesses outside of transactions to be accompanied by memory barriers, and this setup greatly increases the overhead of strong atomicity. Guarding a memory access with a barrier, however, is only necessary if it may conflict with a memory access inside a transaction that may be executed in parallel.[3]

---

[3] Strong atomicity semantics do not cover conflicting memory access outside transactions.

The following Datalog rules decide for a given read- or write-bytecode (outside a transaction) whether it requires a read or write barrier:

```
readInsideTransaction(act, obj:H) :-
                bcGuardedByAtomic(act, readBC), read(act, readBC, obj).
writtenInsideTransaction(act, obj:H) :-
                bcGuardedByAtomic(act, writeBC), write(act, writeBC, obj).
requiresReadBarrier(readBC) :-
                read(act1, readBC, obj),
                writtenInsideTransaction(act2, obj),
                parallel(act1, act2).
requiresWriteBarrier(writeBC) :-
                read(act1, writeBC, obj),
                writtenInsideTransaction(act2, obj),
                parallel(act1, act2).
requiresWriteBarrier(writeBC) :-
                read(act1, writeBC, obj),
                readInsideTransaction(act2, obj),
                parallel(act1, act2).
```

The relation $bcGuardedBy : H \times BC$ is a simple reachability predicate that contains all bytecodes that, starting from the `task` of a given activation object, may be executed inside an `atomic` block.

Modulo the exact points-to analysis used[4], the analysis presented here is almost the same as the optimizations presented by Hindman and Grossman [6]. The difference is the additional clause `parallel(act1, act2)` in each of the above rules. This means that if in the worst case the schedule analysis cannot compute any happens-before relationships (and therefore conservatively classifies all activations as *parallel*) our analysis is equivalent to [6]. If the schedule analysis can compute relevant edges, however, our analysis is more precise allowing the optimizer to remove more read- and/or write barriers.

4.3 Dependence Reduction

Dependence reduction aims at removing →-statements from the source code. This can be beneficial in two ways:

- Removing a →-statement that creates a happens-before relationship between two activations that are already (transitively) ordered can improve the performance of later analyses as well as improve the generated code. Unnecessary transitive →-statements can be found by looking for transitive edges in the schedule.
- Removing a →-statement between two activations that are otherwise not ordered can increase the parallelism in a program.

---

[4] Hindman and Grossman use a points-to analysis that distinguishes objects by type, not by creation site [6].
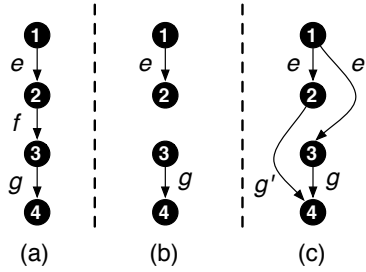
Fig. 6. Fixing the transitive ordering after removing the edge `f`.

Removing a non-transitive edge between two activations may be allowed if the read- and write-sets of both activations are disjoint.

```
requiredEdge(act1, act2) :-
               happensBeforeEdge(act1, act2),
               write(act1, obj),
               readOrWrite(act2, obj).
requiredEdge(act1, act2) :-
               happensBeforeEdge(act1, act2),
               readOrWrite(act1, obj),
               write(act2, obj).
unnecessaryEdge(act1, act2) :- !requiredEdge(act1, act2).
```

When such an edge is removed, however, we must ensure that the transitive ordering is kept intact. Take, for example, the schedule shown in Figure 6(a). If the analysis finds that edge `f` is unnecessary, simply removing it results in the schedule shown in Figure 6(b). This schedule is broken, because by removing `f` the transitive ordering between node 1 and node 3 as well as the transitive ordering between node 2 and node 4 that was present before the removal is missing. After adding the additional edges `e'` and `g'` as shown in Figure 6(c) the transitive ordering is correct again. The parallelism has been increased, however, because activations 2 and 3 can now be executed in parallel. In [3] we present more details about this optimization.

5   Implementation and Future Work

The Datalog parts of our schedule analysis have been implemented and can be found on *http://wiki.github.com/chmaruni/xsched/*. We use the *bddbddb* Datalog system. *bddbddb* is backed by binary decision diagrams (BDDs) and has been shown to scale to large programs using over 10^{14} contexts [18]. The pre-processor and the graph flattening phase are currently under development.

The next step is to integrate our optimizations with a compiler to produce optimized code and to empirically measure how our optimizations compare to the ones from the original papers.

6 Related Work

The *happens-before* ordering was first formulated by Lamport [7] and is the basis of the Java memory model [8]. Despite its significance in the memory model, in Java happens-before edges can be created only implicitly, e.g., by using `synchronized` blocks or `volatile` variables.

The goal of a pointer analysis is to statically determine when two pointer expressions refer to the same memory location. Steengaard [17] and Andersen [2] laid the groundwork for the flow-insensitive analysis of single threaded programs. Because points-to analysis is undecidable in the general case, however, researchers developed a large collection of approximation algorithms specialized for different problem domains [5], including parallel programming.

Rugina and Rinard [14] describe a pointer analysis for programs with structured fork-join style concurrency. For each program point, their algorithm computes a points-to graph that maps each pointer to a set of locations. By capturing the effects of pointer assignments for each thread, their algorithm can compute the interference information between parallel threads. Computing the interference information relies on the lexical scoping of the parallel constructs; it cannot handle unstructured parallelism.

By combining pointer and escape analysis, subsequent projects were able to extend their analyses beyond structured parallelism [15, 11]. Both analyses compute points-to information but do not directly answer as to how two tasks are executed with respect to each other. Further, the tight integration of the pointer analysis with the escape analysis and concurrency analysis is contrary to our goal of separating the concerns of schedule analysis from points-to analysis.

A *may-happen-in-parallel* (MHP) analysis can be used to determine what statements in a program may be executed in parallel [12]. Without flow sensitivity, relating two program statements is of limited use for analyzing programs with unstructured parallelism. If two threads execute the same statements but in different contexts, for example, a context insensitive MHP analysis might unnecessarily classify the statements as parallel. When the programming language is restricted to structured parallelism, as is the case for X10, an intra-procedural MHP analysis can achieve good results, however [1].

Barik [4] describes a context and flow-sensitive may-happen-before analysis that distinguishes threads by their creation site. By using threads as their model, however, they must conservatively assume that a parent thread in the tree runs in parallel with each child thread. In our model a parent activation is known to happen before any child activation because the creation tree is a spanning tree embedded in the schedule.

7 Concluding Remarks

In this paper we showed how an independent schedule analysis can form the basis for different optimizations of parallel programs.

In previous compilers, each optimization had to come with its own model and analysis of concurrent computation. The introduction of an independent

schedule analysis factors out the common aspects of these optimizations, making it easy to not only combine multiple optimizations but also to derive new ones. Combining the optimizations discussed in this paper, for example, allows the optimization of programs that intermix transactional memory with traditional locking. Moreover, the optimization for synchronization removal could be easily adapted to remove atomic sections as well.

The key factor that enabled this approach was a model of parallel computation that allowed a static analysis of the dynamic schedules to be encountered at runtime. Exposing the schedule (and allowing a compiler to analyze and optimize it) is a necessary step in the path towards improving the optimization of parallel programs.

References

1. S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel Analysis of X10 Programs. *In PPoPP, 2007.*
2. L. O. Andersen. Program Analysis and Specialization for the C Programming Language. *Ph.D thesis, DIKU, University of Copenhagen, 1994.*
3. C. M. Angerer and T. R. Gross. Parallel Continuation-Passing Style. *In PESPMA, 2010.*
4. R. Barik. Efficient Computation of May-Happen-in-Parallel Information for Concurrent Java Programs. *In LCPC, 2005.*
5. M. Hind. Pointer Analysis: Haven't We Solved This Problem Yet? *In PASTE, 2001.*
6. B. Hindman, D. Grossman. Strong atomicity for Java without virtual-machine support. *Tech. Rep. UW-CSE- 06-05-01, May 2006.*
7. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM 21, 7 (1978).*
8. J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. *In POPL, 2005.*
9. N. Matsakis, T. Gross. Programming with Intervals. *In LCPC (2009).*
10. S. S. Muchnick. Advanced Compiler Design and Implementation. *Morgan Kaufmann Publishers, 1997*
11. M. G. Nanda and S. Ramesh. Pointer Analysis of Multithreaded Java Programs. *In SAC, 2003.*
12. G. Naumovich, G. Avrunin, and L. Clarke. An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. *In ESEC/FSE-7, 1999.*
13. E. Ruf. Effective Synchronization Removal for Java. *In PLDI, 2000.*
14. R. Rugina and M. Rinard. Pointer Analysis for Structured Parallel Programs. *In TOPLAS, 2003.*
15. A. Salcianu, M. Rinard. Pointer and Escape Analysis for Multithreaded Programs. *In PPoPP, 2001.*
16. T. Shpeisman et al. Enforcing Isolation and Ordering in STM. *In PLDI, 2007.*
17. B. Steensgaard. Points-to Analysis in Almost Linear Time. *In POPL, 1996.*
18. J. Whaley and M. S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. *In PLDI, 2004.*