now happens-before *later* *

Static Schedule Analysis of Fine-grained Parallelism with Explicit Happens-before Relationships

Christoph M. Angerer
ETH Zurich, Switzerland
angererc@inf.ethz.ch

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

Abstract

Current compilers are still largely ignorant of the scheduling of parallel tasks at runtime. Without this information, however, they have difficulties optimizing and verifying concurrent programs.

In this paper, we present a programming model where the program contains explicit scheduling constraints in the form of happens-before relationships between scheduled tasks. This model allows for flexible and fine-grained ad hoc parallelism while still enabling us to statically extract an abstraction of the runtime schedule. The result of this schedule analysis can answer the question as to whether two tasks execute in sequence, exclusively, or in parallel with each other.

Categories and Subject Descriptors D.1.3 [*Software*]: [Concurrent Programming]

General Terms Algorithms, Languages

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 runtime. Absent this knowledge, however, a compiler is missing important optimization and verification opportunities.

In a traditional thread model, the lifetime of a thread and its dependencies on other threads are not stated explicitly; rather, they come about as a side effect of executing low level primitives such as signals and locks. For this reason, it is hard for compilers to construct an approximation of the runtime schedule.

Consider the following short Java method:

* Supported, in part, by the Swiss National Science Foundation grant 200021_120285.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Onward! 2010, October 17–21, 2010, Reno/Tahoe, Nevada, USA.
Copyright © 2010 ACM 978-1-4503-0236-4/10/10...$10.00

```
void begin() {
  this.a = new ThreadA(this);
  this.b = new ThreadB(this);
  a.start();
  //other computations
  b.start();
}
```

Without further information about the lifetime and synchronization of threads a and b, a traditional compiler cannot verify the absence of data races nor can it optimize the parallel code.

One alternative to this unstructured parallelism is to adopt specialized syntactic language features. Systems like OpenMP [20] and Cilk [5, 21], for example, offer lexically scoped fork-join style parallelism in place of ad hoc threads. As a result, these systems are able to better approximate the parallel control flow. Structured parallelism, however, comes at the cost of flexibility, making it difficult to model common patterns such as futures or producer-consumer.

In this paper, we propose a model with explicit task scheduling that keeps the flexibility of threads and enables static reasoning. Given two tasks, a *schedule analysis* can answer the question whether the tasks are sequential, parallel, or exclusive. A compiler can use this information to make parallelism-related decisions during verification and optimization phases.

For representing concurrent programs, we introduce two new primitives to a Java-like language. One primitive schedules a new task and the other explicitly adds a happens-before relationship between two scheduled tasks (Section 3). When executing a program with explicit scheduling, the runtime keeps track of the schedule. The schedule is represented as a graph that exhibits specific structural properties (Section 4). These properties allow us to statically extract an approximation of the runtime schedule (Section 5). We have implemented a prototype and are working on integrating it with an existing Java compiler framework (Section 6).

We build upon a large body of related work (Section 7) for parallel program analysis to design a system that preserves both flexible, unstructured control flow and static analysis of the program schedule. To summarize, this paper makes the following three contributions:

- We define a representation and execution model of parallel programs with explicit happens-before relationships.
- We identify structural properties of abstract schedules.
- We describe an analysis to extract an abstract schedule from a program that was written in or transformed into our representation.

2. The Need for Schedule Analysis

Researchers have developed a wide variety of compiler optimizations and verifications for parallel programs. Adapting such optimizations to parallel programs, however, requires information about what parts of the program might be executed in parallel. The goal of a schedule analysis is to statically compute a mapping $Task \times Task \rightarrow Relation$ to answer the question of how two program tasks relate to each other:

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

Exclusive: Two tasks 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 tasks are neither sequential nor exclusive, they are considered (potentially) parallel.

Generally, the safe and conservative assumption is to over-approximate the 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.

There are numerous examples for optimizations that require or benefit from scheduling information:

Synchronization Elimination aims at removing unnecessary synchronization constructs [22]. A synchronization construct can be removed if all tasks that execute the critical section are sequentially ordered or exclusive.

Region-based Allocation optimizes garbage collection by allocating all or some objects created during a (possibly parallel) computation in a contiguous memory region [24]. The whole region is deallocated as a unit when the computation finishes. To avoid dangling references, however, the compiler must ensure that at the point of de-allocation there are no more parallel tasks that might use the memory.

Polyhedral Analysis tries to automatically introduce parallelism that was not originally specified by the programmer [6]. Parallelism can be increased if, for example, a compiler can show that a happens-before relationship between two tasks can be removed without introducing a data race.

3. Explicit Task Scheduling

Our model is based on lightweight tasks with explicit scheduling. Compared to traditional threads, 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 aPrint = sched(this.print());
  Activation aCompute = sched(this.compute());
  aCompute→aPrint;
}
```

A schedule is represented as a graph of $\langle object, task()\rangle$ pairs. The statement `sched(this.print())`, for example, 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 $\langle o2, t2()\rangle$ is started. The statement `aCompute→aPrint` creates an explicit happens-before relationship between the two activation objects `aCompute` and `aPrint`.

In the code, 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. Therefore, in the example the scheduler implicitly creates two additional edges `now→aCompute` and `now→aPrint`. These edges prevent the immediate execution of the new activations and enable the current task to add additional constraints to the schedule before it finishes.

The above example works, as long as `compute()` does not schedule new subtasks. If it does, however, the schedule would not contain any happens-before edges between the `aPrint` activation and those new activations. Therefore, the scheduler would be allowed to execute `aPrint` before the

subtasks have finished, i.e. too early. However, there is no place where we could create edges to prevent `aPrint` from executing prematurely: In `t()` the subtasks have not yet been created and inside `compute()` we are missing a reference to `aPrint`.

To solve this we can pass the `aPrint` object as a parameter to the `compute()` task and use it to schedule the new subtasks before `aPrint`:

```
task t() {
  Activation aPrint = sched(this.print());
  //pass a reference to aPrint:
  Activation aCompute = sched(this.compute(aPrint));
  aCompute→aPrint;
}

task compute(Activation later) {
  Activation aSubtask =
                sched(this.someSubtask(later));
  //schedule our subtask before later
  aSubtask→later;
}
```

In `compute()` we can pass the reference even further along to `aSubtask`, thus allowing `aSubtask` (and its subtasks, if there are any) to push the execution of the `aPrint` activation further and further into the future until the whole computation is finished. Once the subtasks terminate without inserting new tasks, the scheduler will be able to execute `aPrint`.

3.1 A Recursive Divide-and-Conquer Example

Figure 1 shows an example of a recursive divide-and-conquer algorithm with explicit scheduling. The algorithm sums the elements of an integer array by recursively dividing the array into a left and a right half before computing their sums. The base case of the recursion is reached for sub-arrays of length 1, in which case the sum is trivial.

The class `ArraySum` is an implementation of this algorithm. It defines two tasks: `sum()` divides the work between two children and `subtotal()` adds their results. Figure 2 shows the changes in the schedule when `ArraySum` is started with an array of length 3.

Initially, the schedule contains an activation $\langle o1, sum()\rangle$ for an `ArraySum` object `o1` plus an activation $\langle x, y()\rangle$. $\langle x, y()\rangle$ is provided by the client of `ArraySum` to make use of the result after the computation is finished. A reference to $\langle x, y()\rangle$ is passed to `sum()` through the `later` parameter on line 8. In this schedule, the scheduler can choose $\langle o1, sum()\rangle$ because there are no other outstanding happens-before relationships for that activation.

In the first iteration the array length is greater than 1, which leads to the recursive case starting at line 12. In this branch, we first schedule `this.subtotal()` on line 13. We then add the happens-before relationship `subtotal→later` on line 14, creating an edge between the nodes $\langle o1, subtotal()\rangle$ and $\langle x, y()\rangle$. As shown in Figure 2, this edge defers the execution of $\langle x, y()\rangle$ until the subtotal is available.

```
1  class ArraySum {
2    IntArray arr;
3    int result;
4    Activation left, right;
5
6    ArraySum(IntArray arr) { this.arr = arr; }
7
8    task sum(Activation later) {
9      if(arr.length() == 1) {
10       result = arr.getInt();
11       //end of this task
12     } else {
13       Activation subtotal = sched(this.subtotal());
14       subtotal→later;
15
16       left = new ArraySum(arr.leftHalf());
17       right = new ArraySum(arr.rightHalf());
18
19       sched(left.sum(subtotal))→subtotal;
20       sched(right.sum(subtotal))→subtotal;
21     }
22   }
23
24   task subtotal() {
25     result = ((ArraySum)left.obj()).result
26             + ((ArraySum)right.obj()).result;
27     //end of this task
28   }
29 }
```

Figure 1. Example of a recursive divide-and-conquer algorithm with explicit scheduling.

Lines 16 and 17 split the input array into two halves and pass them to two new instances of `ArraySum`. We store references to both instances in the `left` and `right` fields so that the `subtotal()` task can later read their result.

The subtasks `left.sum()` and `right.sum()` are scheduled on lines 19 and 20. By passing a reference to the `subtotal` activation, the recursive child activations of `sum()` can insert their own `subtotal` activations on line 14, preventing the parent's `subtotal` from executing before the children have finished.

The two scheduling statements add the nodes $\langle o2, sum()\rangle$ and $\langle o3, sum()\rangle$ to the schedule and bind their `later` parameter to the node $\langle o1, subtotal()\rangle$. This is shown in Panel 2 of Figure 2. On the same lines 19 and 20, the two new activations are also scheduled before `subtotal`, thus creating the edges $\langle o2, sum()\rangle \rightarrow \langle o1, subtotal()\rangle$ and $\langle o3, sum()\rangle \rightarrow \langle o1, subtotal()\rangle$.

The scheduler can now choose either $\langle o2, sum()\rangle$ or $\langle o3, sum()\rangle$ for execution. $\langle o2, sum()\rangle$ hits the base case because its array is of length 1. The base case of the recursion on line 9 does not schedule any new tasks and does not create any new happens-before edges, so the schedule

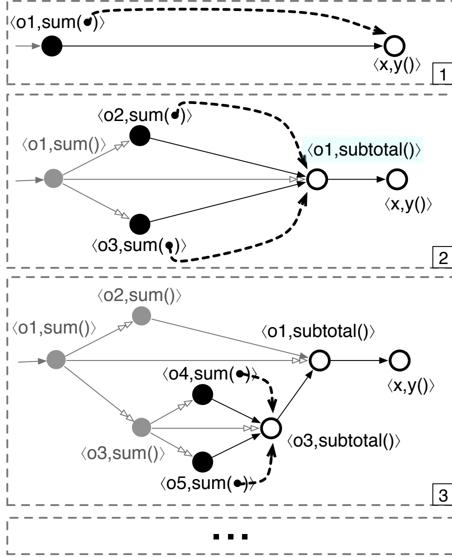

Figure 2. Snapshots of the schedule during execution of the example from Figure 1 for an array of length 3. Satisfied happens-before relationships and already executed nodes are grayed out. Black nodes are eligible for execution. Unfilled nodes have unfulfilled scheduling constraints.

```
task a(Activation later) {
  if(cond) {
    sched(this.b())→later;
  } else {
    sched(this.c())→later;
    sched(this.d())→later;
  }
  sched(this.e())→later;
}
```

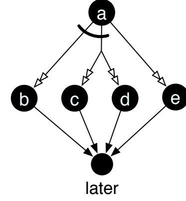

Figure 3. An abstract schedule with a conditional activation. The arc groups the exclusive creation edges.

remains unchanged. The execution of $\langle o3, sum()\rangle$, on the other hand, recursively activates three new tasks: two tasks to compute the partial results and one task to add them.

Panel 3 of Figure 2 shows the state of the schedule after $\langle o2, sum()\rangle$ and $\langle o3, sum()\rangle$ have been executed. The scheduler continues to execute $\langle o4, sum()\rangle$ and $\langle o5, sum()\rangle$ in any order and (because of the original array of length 3) both activations hit the base case and leave the schedule unchanged. Therefore, the schedule can execute the remaining linear chain consisting of $\langle o3, subtotal()\rangle$ and $\langle o1, subtotal()\rangle$ before continuing with the caller at $\langle x, y()\rangle$.

4. Structural Properties of Schedules

This section describes structural properties of schedules.

4.1 Well-formed Schedules

A well-formed schedule guarantees that the scheduler can always choose at least one activation for execution (progress) and that every activation is eventually executed (liveness). Both conditions require that the schedule is a directed acyclic graph. A cycle in the schedule would result in a deadlock where two activations block each other and prevent progress. Assuming that the execution of a task always terminates, an acyclic graph ensures that there is always at least one node that has only incoming edges from already executed activations.

Besides being acyclic, a well-formed schedule also restricts the addition of new happens-before relationships. Because activations can be stored in fields, an activation object may reference an activation that has already been executed. Imagine that one tries to add a happens-before relationship `a1`→`a2` after `a2` has already been executed. The scheduler has no chance of satisfying this edge; since `a2` lies in the past, the scheduler cannot retroactively execute anything before that.

To prevent such unresolvable scheduling conflicts, the scheduler allows an edge `a1`→`a2` to be added from within an activation `a0` only if there exists an edge `a0`→`a2`. This edge ensures that `a2` is scheduled after `a0` and therefore is still unexecuted at the time of the edge creation. It is not necessary to require an edge `a0`→`a1`, however, because it is not a problem if the source of a happens before edge has already been executed.

4.2 Conditional Activation

Conditional control flow can result in a conditional activation of a task. The `if` statement on line 9 of Figure 1, for example, results in three activations in the `else` branch but none on the `true` branch.

At compile-time, the analysis generally cannot determine which branch is executed at runtime. Therefore, all possible executions must be taken into account. Two activations that are created in different branches of a conditional statement are *exclusive* because they cannot co-exist in the same run of the program. Figure 3 shows an example where node `b` is exclusive to both `c` and `d` but parallel to `e`. Nodes `c` and `d` are parallel to each other as well as parallel to `e`. Graphically, exclusive edges are connected by arcs.

4.3 Creation Tree

The scheduler implicitly adds an edge between the current activation `now` and all the tasks it schedules. Those initial edges are called *creation edges*. We depict creation edges with double arrow heads. Because an activation has exactly one creator, the creation edges form a *spanning tree* that is embedded into the schedule.

The creation tree is a fundamental data structure that enables many of the operations needed during the analysis. Its importance comes from two basic properties:

1. If one activation x is the direct or indirect parent of another activation y in the creation tree, it is guaranteed that x always executes before y because x creates y.

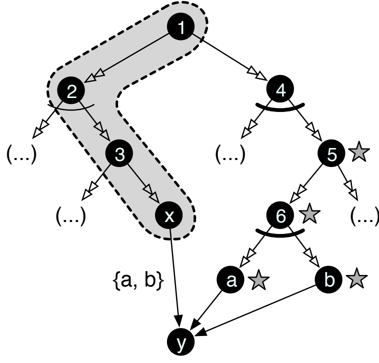

Figure 4. Marking nodes in the creation tree to test if $x \to y$ is genuine. The fence is circled by the dotted line, marks are shown as stars.

2. If $x \twoheadrightarrow y$ is a creation edge, the existence of the child activation y implies the existence of its parent x, written $y \Rightarrow x$. In fact, this relationship is transitive, thus implying the existence of all parents up to the root of the creation tree (the initial activation that started the program). The inverse, however, is not true. Due to conditional activations, one cannot deduce the existence of a child activation simply from the existence of the parent.

4.4 Genuine Edges

The analysis uses the schedule and its embedded creation tree to decide how two activations relate to each other.

Sequential Activations: If two nodes are connected by a path in the schedule, their execution is ordered and therefore sequential. If there is no such path, the activations are either exclusive or parallel.

Exclusive Activations: If we find that in the creation tree two nodes are connected to their common ancestor by conditional edges, the two activations are exclusive.

Parallel Activations: If two nodes are connected to their common ancestor by parallel edges, the activations are considered parallel.

Because happens-before relationships can be created conditionally, many parts of the analysis only consider genuine edges. A genuine edge $x \to y$ is an edge where the existence of the source node x implies that the edge exists, denoted $x \Rightarrow x \to y$. Genuine edges are useful because we know that if the node x executed at all, it executed before y. Edges that are not genuine are ignored by the analysis, thus over-approximating the parallelism.

To determine whether $x \Rightarrow x \to y$ we record all the activations $creators(x \to y)$ that unconditionally create the edge $x \to y$. We can now rephrase the problem to check if the existence of x implies the existence of at least one node c that creates the edge: $\exists c.x \Rightarrow c \land c \in creators(x \to y)$. This predicate can be approximated using the creation tree.

Figure 4 shows an example of a creation tree. We want to compute whether the edge $x \to y$ is genuine. The algorithm starts by marking the *fence*. The fence is comprised of all the nodes from the edge source up to the root of the creation tree. As described earlier, the existence of the fence nodes is implied by the existence of x.

The goal is now, to check whether any node in the fence implies the existence of any node that creates the edge $x \to y$. We do so by iteratively marking nodes, walking up the creation tree, until we either mark a node in the fence, in which case the edge is genuine, or no more nodes can be marked, in which case the edge is not genuine.

The label on edge $x \to y$ in Figure 4 indicates that the edge was created by activations a and b. Therefore, the algorithm initially marks the nodes a and b and continues with node 6 as the parent of a and b. Because *all* its exclusive children were marked, and thus all possible execution paths are covered, node 6 can be marked as well. The mark on 6 is sufficient to further mark node 5 because node 6 was created unconditionally.

In the example, node 4 cannot be marked because there is a conditional unmarked sibling of node 5. Therefore, there is a program execution that will create node x but not nodes a or b and thus not the edge $x \to y$. This concludes that $x \to y$ is not genuine.

If the program was modified to create node 5 unconditionally, the algorithm would eventually mark the fence in node 1, showing that $x \to y$ were genuine.

4.5 Recursion

Example 1 contains a recursive activation of the task `sum()` on lines 19 and 20. It is important to detect recursion to prevent infinite expansion of the creation tree during analysis.

In our framework, a recursion is detected as soon as an activation of a task `t()` directly or indirectly causes the creation of another activation of the same task `t()`, but possibly with different `this`-objects. Multiple occurrences of the same task `t()` on an execution trace are not automatically considered recursive, however. It is necessary that the recursion is "self-induced"; the second activation of `t()` must be a result from the first activation of `t()`.

For example, a task `s()` could schedule `o1.t()` and `o2.t()` and add a happens-before constraint between the two. In this case, the execution of `o2.t()` is not considered recursive even though it is executed after `o1.t()` because it did not cause the activation of `o1.t()`.

Given a node in the schedule, we can test for recursive activation by walking up the creation tree. If the node represents a recursive activation of a task `t()`, the creation tree will contain a parent with the same task `t()`.

The analysis records recursive activations in the abstract schedule by adding *recursion edges*. Recursion edges are creation edges that are treated specially when testing whether two nodes are exclusive or parallel.

Imagine an activation a that conditionally creates two activations b and c. In the non-recursive case, b and c are considered to be exclusive. If there is a recursion around a, however, a might create b in the first iteration and c in the second iteration. Because the analysis cannot distinguish the individual iterations, it must assume that b and c are parallel.

5. Schedule Analysis

At the core of the schedule-driven analysis is a standard points-to analysis for object-oriented programs such as the analyses presented in [25] or [23]. The points-to information is necessary because we need information about the target object when a task is scheduled. Similarly, because activations are first-class objects, we need the pointer information to compute the sources and targets of new happens-before edges.

A points-to analysis is driven by the control-flow graph. That is, the order in which nodes are visited and the paths of the information flow are determined by the edges of the CFG. In our schedule-driven analysis, the schedule graph augments this (inter-procedural) role of the CFG for guiding the analysis of parallel constructs. During analysis, the schedule determines in what order the nodes are visited and how information flows between them.

The interface between the points-to analysis and the schedule analysis is an *abstract heap*: a data structure containing the points-to information. For the schedule analysis, the abstract heap is a largely opaque data structure that is defined by the points-to analysis at hand. The schedule analysis requires methods for merging heaps and for querying a heap to determine the points-to set for a given variable.

The analysis works by visiting each node in the abstract schedule until a fixed point has been reached. Analyzing a single node is done in three steps:

1. A heap abstraction is computed by combining the heaps flowing into the node through the incoming edges.
2. The combined heap functions as the input to an incremental pointer analysis; the result of this analysis is an updated heap containing the new points-to information.
3. For non-recursive nodes, the points-to information is used to find newly created activations and/or happens-before edges and to incorporate them into the abstract schedule. If the current node is a recursive activation, however, we instead add a recursion edge that feeds back the result heap and re-open the parent for analysis until a fixed point has been reached.

5.1 Combining Incoming Heaps

The abstract heap at the beginning of an activation a must approximate the effects of all the activations that, at runtime, could execute before a. For sequential executions, the execution order is captured by the happens-before relationships in the abstract schedule. The question is, therefore, what influence can parallel activations have on the initial heap of a?

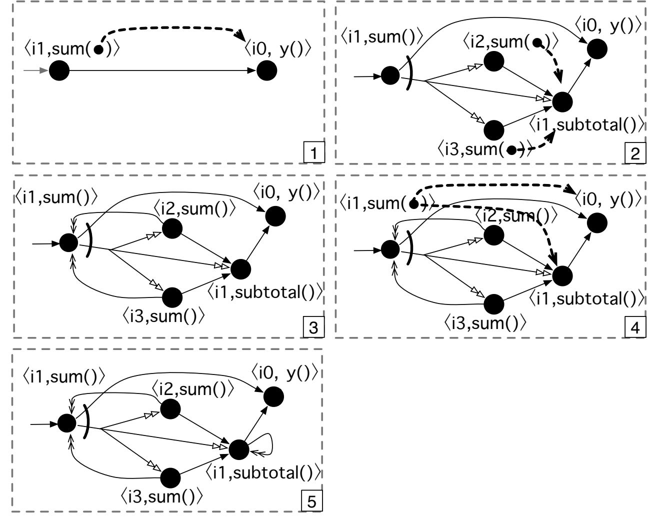

Figure 5. Schedule analysis of Example 1.

As it turns out, parallel activations can be ignored if data races are considered to be illegal and if we are willing to 'lazily' detect a data race not in the activations that are directly involved but only in one of their successors. This lazy approach is in contrast to other analyses that iteratively reanalyze threads and update their interference information until a fixed point is reached. For the rare cases where data races should be deliberately allowed we can fall back to such an iterative algorithm, however.

Imagine a special node `exit` that marks the end of the program and happens after all other activations. Then there is always at least one node c for any two unordered activations a and b that happens after a and b: $a \to^* c \wedge b \to^* c$. Such a join node c is the point where a data race detection can find concurrent accesses to the same memory locations by comparing the read/write sets of the parallel activations.

Therefore, we can simply assume the absence of data races between a and b because this assumption will be verified later in c. If there is no data race, two activations cannot interfere with each other. For this reason, while analyzing a we can ignore all unordered activations b, c, ... and derive the abstract heap by combining only the heaps of the predecessors connected to a by incoming edges.

5.2 Locks

Synchronizing an arbitrary number of concurrent tasks requires an additional synchronization primitive such as mutexes, atomic compare-and-swap, or locks [11, p.37ff]. Inter-task synchronization requires the analysis to compute the fixed point for the implicit information flow between synchronization points. For space reasons, we do not discuss synchronization primitives but the analysis can be extended to handle them.

5.3 Analysis of the Example

Figure 5 shows how the schedule analysis proceeds for the example of Figure 1. The analysis is at the point where a client has scheduled the `sum()` task with the abstract object `i1`. The client further activated $\langle i0, y() \rangle$ to make use of the result later. The relevant part of the abstract schedule is shown in Panel 1 of Figure 5.

The analysis can start to analyze node $\langle i1, sum() \rangle$ because all the preceding nodes have been analyzed. The first step of computing the initial heap is trivial, because there is only one incoming edge. Therefore, we can immediately start the points-to analysis.

Because this was the first time the points-to analysis was started for this node, the resulting abstract heap will contain the three new activations that were created on lines 13, 19, and 20 of Figure 1. Panel 2 of Figure 5 shows the updated abstract schedule after adding the new activation nodes and the corresponding happens-before edges. It is also shown that the `later` parameters of the two `sum()` activations are both bound to the same $\langle i1, subtotal() \rangle$ node.

This finishes the analysis of the first node and the analysis consults the schedule to see which node to analyze next. The options are $\langle i2, sum() \rangle$ and $\langle i3, sum() \rangle$ because they are the only nodes that have no unanalyzed predecessors. There is only one incoming edge to either node and the points-to analysis can be started immediately. Because for both nodes we detect a recursion around `sum()`, we add recursion edges as shown in Panel 3. This re-opens $\langle i1, sum() \rangle$ for analysis.

Back at node $\langle i1, sum() \rangle$, the recursion edges require the analysis to merge the `later` parameter of the `sum()` tasks. As shown in Panel 4, due to the recursion, `later` may point to $\langle i1, subtotal() \rangle$ or $\langle i0, y() \rangle$.

There is more than one incoming edge to $\langle i1, sum() \rangle$; thus we must combine the incoming heaps before we can start the points-to analysis. Looking at the creation tree reveals that $\langle i2, sum() \rangle$ and $\langle i3, sum() \rangle$ are parallel. A data race detection can verify that both tasks can safely run in parallel because both activations access disjoint regions of the array.

Having merged the heaps, the pointer analysis can be restarted for node $\langle i1, sum() \rangle$. In this example, the heap before the pointer analysis is equal to the heap returned by the pointer analysis and we have found a fixed point.

There were no new nodes created, but checking for newly created edges reveals that the statement `subtotal`→`later` on line 14 results in an additional recursion edge from $\langle i1, subtotal() \rangle$ to itself because both variables `subtotal` and `later` may point to the same $\langle i1, subtotal() \rangle$. This loop represents the chain of `subtotal()` activations that can occur at runtime.

Panel 5 of Figure 5 shows the state of the abstract schedule after the `sum()` recursion has been analyzed. The analysis proceeds with computing the fixed point for the recursive node $\langle i1, subtotal() \rangle$ before finishing with node $\langle i0, y() \rangle$.

6. Prototype

We have implemented a prototype of the schedule analysis. The prototype works on a simplified object-oriented language and can be found at `http://github.com/chmaruni/XSched`. We are now in the process of integrating the analysis with the WALA Java analysis library [28].

7. Related Work

The *happens-before* ordering was first formulated by Lamport [14] and is the basis of the Java memory model [15]. Despite its significance in the memory model, in Java happens-before edges can be created only implicitly, for example 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 [26] 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 [12], including parallel programming.

Rugina and Rinard [23] 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 [18, 24]. 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 [19]. 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 has been done for the X10 programming language [1], an intra-procedural MHP analysis can achieve good results, however.

Barik [3] describes a context and flow-sensitive may-happen-before analysis that distinguishes threads by their creation site. Barik introduces a ‘thread creation tree’, which is closely related to our creation tree. 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.

As an alternative to data-flow analysis, many systems apply techniques based on type theory and related formalisms for analyzing parallel programs. Among the many approaches used are typestates [4], ownership types [7], effect systems [9, 16], and access permissions [27].

Actor-based systems [10, 13] avoid many synchronization issues by removing the need for a global schedule altogether. Actors are entities that communicate asynchronously by sending and receiving messages to and from each other. There is no restriction on the order in which messages arrive and an actor has no direct control over the message passing mechanism. This lack of synchronization requires the actor model to avoid mutable shared state whereas our work is based on a shared-memory model. Process calculi, such as the join-calculus [8] and π-calculus [17], permit formal reasoning about systems with autonomous entities.

8. Concluding Remarks

Fully utilizing the increasing number of cores in modern processors requires finer- and finer-grained parallelism. Fine-grained parallelism is characterized by small tasks with only short pieces of sequential code. Many powerful compiler optimizations for single-threaded code, however, become ineffective when the sequential parts are too short. At the same time, new parallelism-aware optimizations require knowledge about the task scheduling at runtime, but this information is not available in current compilers.

Instead of each project inventing its own model of concurrency, we propose an independent discipline of schedule analysis. From this, we expect the same beneficial synergies for future parallel optimizations as with the theory of points-to analysis, which allowed optimizations to focus on their optimization problems instead of computing points-to sets.

We believe that static schedule analysis is a necessary step towards efficient next-generation compilers for multi-core systems.

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] R. Barik. Efficient Computation of May-Happen-in-Parallel Information for Concurrent Java Programs. *In LNCS, 2006.*

[4] N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying Correct Usage of Atomic Blocks and Typestate. *In OOPSLA, 2008.*

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *In PPoPP, 1995.*

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *In PLDI, 2008.*

[7] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. *In OOPSLA, 1998.*

[8] C. Fournet and G. Gonthier, The Join-Calculus: A Language for Distributed Mobile Programming. *In APPSEM, 2000.*

[9] A. Greenhouse and J. Boyland An Object-Oriented Effects System. *In ECOOP, 1999.*

[10] A. Gul. An Overview of Actor Languages. *In SIGPLAN Not., vol. 21, pp.58–67, 1986.*

[11] M. Herlihy and N. Shavit. The Art of Multiprocessor Programming. *Morgan Kaufmann, MA-USA, 2008.*

[12] M. Hind. Pointer Analysis: Haven't We Solved This Problem Yet? *In PASTE, 2001.*

[13] R. K. Karmani, A. Shali, and A. Gul. Actor Frameworks for the JVM Platform: A Comparative Analysis. *In PPPJ, 2009.*

[14] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM 21, 7 (1978).*

[15] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. *In POPL, 2005.*

[16] N. Matsakis and T. Gross. Reflective Parallel Programming. *Int HotPar, 2010.*

[17] R. Milner. Communicating and Mobile Systems: The π-Calculus. *Cambridge University Press, NY, USA, 1999.*

[18] M. G. Nanda and S. Ramesh. Pointer Analysis of Multi-threaded Java Programs. *In SAC, 2003.*

[19] G. Naumovich, G. Avrunin, and L. Clarke. An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. *In ESEC/FSE-7, 1999.*

[20] OpenMP Specification: Version 3.0. *http://openmp.org/.*

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

[22] E. Ruf. Effective Synchronization Removal for Java. *In PLDI, 2000.*

[23] R. Rugina and M. Rinard. Pointer Analysis for Structured Parallel Programs. *In TOPLAS, 2003.*

[24] A. Salcianu, M. Rinard. Pointer and Escape Analysis for Multithreaded Programs. *In PPoPP, 2001.*

[25] M. Sridharan and R. Bodík. Refinement-based Context-sensitive Points-to Analysis for Java. *In PLDI, 2006.*

[26] B. Steensgaard. Points-to Analysis in Almost Linear Time. *In POPL, 1996.*

[27] S. Stork, P. Marques, and J. Aldrich. Concurrency by Default: Using Permissions to Express Dataflow in Stateful Programs. *In OOPSLA, 2009.*

[28] T.J. Watson Libraries for Analysis (WALA). *http://wala.sourceforge.net*