# OoOJava: Software Out-of-Order Execution

## ABSTRACT
Current tools make developing parallel software difficult. Even experts commonly introduce both race conditions and deadlocks into parallel software. We present OoOJava, a new approach to parallel programming inspired by out-of-order hardware. Out-of-order processors have long extracted unstructured parallelism from sequential instruction streams. In our approach, a developer annotates code blocks as reorderable tasks to decouple these blocks from the parent execution thread. OoOJava extract all data dependences through static analysis to generate an executable that is guaranteed to preserve the behavior of the original sequential program.

We have implemented OoOJava and achieved speedups ranging from **[[XXX×]]** to **[[YYY×]]** on our **[[five]]** benchmarks. The combination of a simple parallelism model, compiler feedback, and speedups are indications that out-of-order execution-based programming models can become mainstream.

## 1. INTRODUCTION
Mainstream processors have been released with as many as twelve cores and processors with many, many more cores will become commonplace within a few years. Current software tools lag behind hardware platforms; developing parallel software using these tools is both difficult and error-prone. Today's tools require developers to reason carefully about the interactions of many parallel threads to write correct software — a task which even the best developers find extremely difficult. Experience shows that application written using existing models are prone to both races and deadlocks.

Deterministic programming models have been recognized to simplify developing parallel code by eliminating race conditions [6]. Deterministic parallel programming systems exist today that strive to take sequential code with parallelization annotations and generate a parallel implementation with the same behavior but better performance. However, much of this work either limits the structure of the code that can be parallelized (loops only) [11], constrains the usage of data structures [16, 4], requires extensive annotations [21, 7], or only guarantees determinism under unchecked conditions (disjoint data structures) [25, 12, 19].

Hardware has long extracted unstructured parallelism from sequential instruction streams through out-of-order execution [26]. Processors dynamically extract dependences between sequential instructions and then execute the instructions out-of-order while preserving the dependences. This paper leverages the same proven techniques at a coarser granularity to parallelize software.

OoOJava is a compiler-assisted approach that leverages developer annotations along with static analysis to provide a deterministic parallel programming model. OoOJava extends sequential Java with a single annotation, a task, to instruct the compiler to consider a code block for out-of-order execution. OoOJava executes tasks as soon as their data dependences are resolved and guarantees that the execution of an annotated program preserves the exact semantics of the *serial elision*, the sequential program obtained by removing all annotations. Therefore annotations never affect the program's correctness, but merely its performance.

This basic approach has been known for some time. We leverage a new extension to pointer analysis, disjoint reachability analysis, to augment our modified approach to effects [14]. OoOJava uses the results of the static disjoint reachability analysis to generate a handful of lightweight comparisons that allow it to safely dynamically extract parallelism even when the heap accesses cannot be statically determined to be disjoint. We combine this with a new value forwarding approach that is analogous to register renaming and eliminates write-after-write and write-after-read hazards. Together, these techniques allow OoOJava to parallelize a wider-range of programs and require fewer changes to sequential code than previous systems.

This paper makes the following contributions:

- **OoOJava:** It presents a deterministic parallel programming model that extends Java with the task annotation while preserving the program's sequential semantics. Because OoOJava makes few changes to Java, developers will likely find it easy to use.
- **Dependence Analysis:** OoOJava uses static analysis to discover data dependences.
- **Software-Based Out-of-Order Execution:** Processors execute instructions out-of-order to extract fine-grained, unstructured parallelism. OoOJava adapts out-of-order execution techniques in software to parallelize code blocks and guarantees that the execution respects all dependences.
- **An Implementation and Evaluation:** We have implemented OoOJava and have evaluated its performance on **[[ten]]** benchmarks.

The remainder of the paper is organized as follows. Section 2 presents an example. Section 3 presents an overview of our approach. Section 4 presents our approach to managing variable dependences. Section 5 presents our approach to managing heap dependences. Section 6 discusses various aspects of our system. Section 7 evaluates the approach on several benchmark applications. Section 8 discusses related work; we conclude in Section 9.

## 2. EXAMPLE
We present a compiler example to illustrate OoOJava. Figure 1 presents the `typeCheckAndFlatten` method that iterates over a set of methods and performs the initial stages of compiling those methods. Consider this method without the `task` keyword or the associated curly braces. The loop retrieves the abstract syntax tree (AST) associated with a method in Line 5, type checks the AST in Line 7, and finally flattens the AST into a control flow graph in Line 8. If different loop iterations operate on different AST's and the AST's do not share mutable data, Lines 7 through 8 from different loop iterations can be run in parallel.

However, the loop iterations are not completely independent. Line 11 stores the control flow graph for every loop iteration into the same hashtable and therefore has a dependence on itself from the previous loop iteration.

OoOJava extends the sequential programming model with the task annotation, that hints to the compiler that a block of code can be decoupled from the parent thread. A task is declared with the keyword `task`, followed by a name for the task and a pair of braces enclosing the block. Task names have no semantic meaning but are useful for communicating feedback about dependences between tasks to the developer. Tasks do not introduce a new variable scope — variables declared in a task can be accessed outside of the task.

```
1  public void typeCheckAndFlatten() {
2    Iterator<MethodDesc> methodsItr=allMethods.iterator();
3    while( methodsItr.hasNext() ) {
4      MethodDesc m = methodsItr.next();
5      AST ast = m2ast.get( m );
6      task tpar {
7        ast.typeCheck();
8        CFG cfg = ast.flatten();
9      }
10     task tser {
11       m2cfg.put( m, cfg );
12     }
13   }
14   m2cfg.serializeToDisk();
15 }
```

**Figure 1: Compiler Example**

Task must have a single exit.

In our example, we enclosed Lines 7-8 with the task `tpar` to decouple their execution from the main thread. The task `tpar` can be safely parallelized if the object referenced by variable `ast` is unique from the previous iterations, and the objects in the AST that are updated are only from one instance of an `AST` object. Note that parallelizing this loop poses a problem for most automatic parallelization systems — it is extremely difficult to statically determine that different loop iterations access different `AST` objects.

The statement at Line 11 poses a challenge for parallelization — it has a dependence on the result of flattening the AST and a sequential dependence on itself from the previous loop iteration. We enclose Line 11 in second task named `tser` to allow the main thread to continue past this statement and to separate it from the parallel work in task `tpar`. The sequential dependence will force all instances of `tser` to execute serially.

It is important to note that OoOJava guarantees that the annotated program always preserves the semantics of its serial elision, the sequential program obtained by removing all annotations. Annotations in OoOJava never effect the program's semantics, but merely its performance.

# 3. OVERVIEW

The execution model for OoOJava is inspired by out-of-order processors. An out-of-order processor takes as input an instruction stream. When the processor *issues* the next instruction in the stream, it records any dependences of the instruction on previously issued instructions. When an instruction's dependences are resolved and the required functional unit is available, the instruction is *dispatched* to the functional unit. When completed, the instruction *retires* by updating the processor state.

Similarly, when a thread reaches the definition of a task, it allocates a record for the task and makes a runtime count of all outstanding dependences the task has on its sibling tasks. The thread then issues the task and then skips to the end of the task declaration to immediately continue to execute its own code. When the task's dependences are fully resolved, it is dispatched. When the task finishes execution, it waits for all of its children to retire before it retires.

Tasks can be nested and there is an implicit top-level task for the main method; an important property of OoOJava is that tasks form a tree at runtime. A parent is responsible for issuing children and managing the dependences among its children and itself, therefore tasks in different subtrees can proceed concurrently on different processing cores and parallelize the scheduling load.

An important component to our approach is conservatively extracting dependences between tasks to ensure that parallel execution preserves the behavior of the program's serial elision. Program dependences can take two forms: control dependences and data dependences. Control dependences are handled by OoOJava implicitly by constraining a task to have a single exit. One implication is that a task cannot throw an exception that it does not catch. If a task throws an exception that it does not catch, the entire program is terminated. OoOJava does not handle such exceptions precisely — it is likely that OoOJava will have already started executing code after the failed task.

A task may have a data dependence on another task through a variable or through conflicting heap accesses to an object. Different runtime instances of the same task use the same variables. Like register renaming in out-of-order hardware, a key component of parallelization is to eliminate write-after-write and write-after-read hazards on variables by forwarding values directly to the consuming task. Section 4 presents our compiler analysis for extracting variable dependences and our runtime strategy for respecting them.

OoOJava structures task dependence relations to minimize overheads. Nested task annotations can generate dynamic trees of tasks. If we allow dependences between arbitrary tasks in the hierarchy then the maintenance of dependences becomes a global problem and a potential bottleneck. OoOJava restricts task dependence relations to be only parents-child or sibling-sibling in nature. OoOJava enforces this structure by attributing a child's dependences to its parent and only retiring tasks after all of their children have retired. This structure simplifies the management of dependences by localizing the problem; a parent manages the dependences between itself and its children. Moreover, it allows our implementation to parallelize dependence tracking and therefore support scaling to systems with a large number of cores.

# 4. VARIABLE DEPENDENCES

A task has a variable dependence on a second task if it reads a variable that was written to by the second task. In the example presented in Section 2, task `tser` reads the value of variable `cfg` that the most recent instance of task `tpar` wrote. Therefore, the task `tser` has a variable dependence on the most recent instance of the task `tpar` for the value of the variable `cfg`. The variable dependence analysis statically extracts a conservative set of variable dependences between tasks.

## 4.1 Variable Dependence Analysis

Variable dependence analysis abstracts the source of a variable's current value with a variable source tuple. A variable source tuple contains three parts: (1) the name of the task that produced the value, (2) which instance of that task, relative to the most recent dynamic instance, produced the value, and (3) which variable it wrote the value to. Variable source tuples are useful as they statically characterize how values in variables are propagated between tasks.

To be more precise, the variable source for a variable $v_1$ has the form $\langle t, a, v_2 \rangle$. The combination of $t \in T$ and $a$ together statically specify a dynamic instance of a task: $t$ is a task and $a$ is the age of that task (i.e. how many instances of that task have been issued between the source instance and the current program point). To bound the analysis, $a$ is taken from $\{0, 1, \ldots, k\}$, where a variable source with $a = k$ means an instance with an unknown age. The variable $v_2 \in V$ specifies which variable at the exit of the task given by $t$ and $a$ contains the relevant value. A variable source serves as a complete abstract address for the value of a variable.

We define in-set variables and out-set variables as useful concepts for our dependence analysis. A task's in-set variables are variables that an task may read values from that were written to before the execution of that task. Similarly, out-set variables are variables a task write values to that may be read outside of that task. Our analysis calculates in-set and out-set variables by examining variable sources at a task's enter and exit points. In the example, task `tpar` has the in-set $\{\texttt{ast}\}$ and the out-set $\{\texttt{cfg}\}$ while task `tser` has the in-set $\{\texttt{m2cfg}, \texttt{m}, \texttt{cfg}\}$.

**[[note that variable sources + out-sets + in-sets tell us how to route values between tasks]]**

### 4.1.1 *Abstract Domains*

The analysis is structured as a standard forward dataflow analysis. The set $t \in T$ contains all tasks in the program. The set $a \in A = \{0, \ldots, k\}$ is the domain of variable source ages, where $k$ is interpreted as the top value (i.e. unknown age). The set $V$ contains all variables in the program and $L$ maps program points to live variables **[[clarify in vs out]]**. We define the domain of variable source tuples as follows: $\langle t, a, v\rangle \in S \subseteq T \times A \times V$. The analysis computes a mapping $M \subseteq V \times S$ at each program point. The relation $M$ maps a variable to the set of variable source tokens that describe the source of the variable's value. At program entry, $M$ is empty.

The mapping $M$ forms a lattice. The partial order ($\sqsubseteq$) is defined by the subset relation ($\subseteq$); join($\sqcup$) is set union ($\cup$); bottom($\perp$) is the empty set($\emptyset$); and top($\top$) is the maximally full set. The lattice $M$ is of finite height because both the set of live variables and the set of variable source tuples are finite.

### 4.1.2 *Program Representation*

To simplify the transfer functions we decompose program statements into simple operations: variable reads, variable writes, and variable copy statements. For example, method invocations are viewed as a read of every argument and a write to the return value. A read from variable `x` is denoted by $\mathrm{rd}(\texttt{x})$, likewise a write to `x` is $\mathrm{wr}(\texttt{x})$. The analysis recognizes the variable copy statement `x=y` as a special case. Finally, the entry and exit points of tasks are relevant; for task `t` the analysis recognizes $\texttt{enter}(t)$ and $\texttt{exit}(t)$.

For a statement $s$ in a method's control flow graph, we define the program point just before $s$ as $\bullet s$ and the program point just after $s$ as $s\bullet$.

### 4.1.3 *Transfer Functions*

All transfer functions have the form $M' = (M - \text{KILL}) \cup \text{GEN}$. We define the convenience function $M(x) = \{\langle x, s\rangle \in M\}$. Figure 2 presents the transfer functions for the variable source analysis.

**Write Statement:** When a statement writes to a variable the enclosing task becomes the new source of that variable's value. The GEN set for the statement $\mathrm{wr}(\texttt{x})$ creates the tuple $\langle x, \langle t, 0, x\rangle\rangle$, indicating that the current value of `x` was written to by most recent instance of the task $t$ and was stored in the variable `x` at the exit of task $t$. Writing a new value to a variable removes the old values of that variable. Therefore, the KILL set for the statement $\mathrm{wr}(\texttt{x})$ removes all of the previous sources for the variable `x`.

**Read Statement:** Our compiler implements several optimization to reduce the overhead of communicating values through variables from one tasks to another. The transfer functions for statements that read from variables must take these optimizations into account. If a statement reads from a variable whose value was written by a child task of the current task, the statement must wait (stall) for the child task to complete and copy the value from the child. If the compiler can statically determine the exact child task the statement reads from, it can optimize the code to eliminate future dynamic checks by copying the values for all other variables that it can also determine are written to by the same task. The single child source case gives the transfer function for this case. The KILL part of the transfer function for the single child source for $\mathrm{rd}(\texttt{x})$ kills the variable sources for all variables with the same task source as the variable `x`. The GEN part of the transfer function replaces the variable sources for these variables with a new source for the current task to indicate that the variables' values are immediately available in the variables.

| `st` | $M' = (M - \text{KILL}) \cup \text{GEN}$ |
|---|---|
| $\mathrm{wr}(\texttt{x})$ | $\text{KILL} = \{\texttt{x}\} \times M(\texttt{x})$<br>$\text{GEN} = \{\langle \texttt{x}, \langle t_{\text{curr}}, 0, \texttt{x}\rangle\rangle\}$ |
| $\mathrm{rd}(\texttt{x})$ | Single child sources:<br>If $M(\texttt{x}) = \{\langle t_{\text{child}}, a, v_1\rangle, \ldots, \langle t_{\text{child}}, a, v_m\rangle\} \wedge$<br>$t_{\text{child}}$ is a child of $t_{\text{curr}}$,<br>$Y = \{\forall y \in L \mid \exists w_1, \ldots, w_k,$<br>$M(y) = \{\langle t_{\text{child}}, a, w_1\rangle, \ldots,$<br>$\langle t_{\text{child}}, a, w_k\rangle\}\}$<br>$\text{KILL} = \{\{y\} \times M(y) \mid \forall y \in Y\}$<br>$\text{GEN} = \{\{y\} \times \{\langle t_{\text{curr}}, 0, y\rangle\} \mid \forall y \in Y\}$. |
| | Mixed sources case:<br>If $M(\texttt{x}) = \{\langle t_{\text{child}}, a, v_1\rangle, \langle t, a', v'\rangle, \ldots\} \wedge$<br>$t_{\text{child}}$ is a child of $t_{\text{curr}} \wedge t \neq t_{\text{child}}$,<br>$\text{KILL} = \{\texttt{x}\} \times M(\texttt{x})$,<br>$\text{GEN} = \{\texttt{x}\} \times \{\langle t_{\text{curr}}, 0, \texttt{x}\rangle\}$. |
| | No child sources case:<br>$\text{KILL} = \text{GEN} = \emptyset$ |
| $\texttt{x} = \texttt{y}$ | $\text{KILL} = \{\texttt{x}\} \times M(\texttt{x})$<br>$\text{GEN} = \{\langle t, a, v\rangle \in M(\texttt{x}) \mid t$ is a child $t_{\text{curr}}\} \cup$<br>$\{\langle t_{\text{curr}}, 0, \texttt{x}\rangle \mid \langle t, a, v\rangle \in M(\texttt{x}) \wedge$<br>$\neg t$ is a child $t_{\text{curr}}\}$ |
| $\texttt{enter}(t_{\text{curr}})$ | $\text{KILL} = \{\langle t, a, w\rangle \mid \forall \langle t, a, w\rangle \in M \wedge t = t_{\text{curr}}\}$<br>$\text{GEN} = \{\langle t, a \oplus 1, w\rangle \mid \forall \langle t, a, w\rangle \in M \wedge$<br>$t = t_{\text{curr}}\}$<br>$a_1 \oplus a_2 = \begin{cases} a_1 + a_2 & \text{if } a_1 + a_2 < k, \\ k & \text{otherwise.} \end{cases}$ |
| $\texttt{exit}(t_{\text{curr}})$ | $Z = \{z \in L \mid M(z) = \{\langle t_{\text{child}}, a, v\rangle, \ldots\} \wedge$<br>$t_{\text{child}}$ is a child of $t_{\text{curr}}\}$<br>$\text{KILL} = \{\{z\} \times M(z) \mid z \in Z\}$<br>$\text{GEN} = \{\langle z, \langle t_{\text{curr}}, 0, z\rangle\rangle \mid z \in Z\}$. |

**Figure 2: Transfer functions for variable dependence analysis, where $t_{\text{curr}}$ is the task currently executing and age $0$ names the most recent instance of $t_{\text{curr}}$.**

If the compiler determines that a variable may contain a value written to by a child task but cannot statically determine the exact task it can only copy the value for the current variable. The mixed source case gives the transfer function for this case. The KILL part of the transfer function for the statement $\mathrm{rd}(\texttt{x})$ removes the old variable sources for the variable `x`. The GEN part of the transfer function for the statement $\mathrm{rd}(\texttt{x})$ adds a new variable source for the current task to indicate that the variable's values are immediately available in the variable.

If none of the sources for a variable's value is a child, then the value is immediately available and the read has no effect. The transfer function for this case does not change $M$.

**Copy Statement:** The variable dependence analysis uses a separate transfer function for the variable copy statement `x = y` rather than decomposing it into the combination of a read and a write statement, because the semantics of a variable copy can be utilized by the analysis. The key observation is that copy statements can be evaluated lazily and therefore do not immediately need the value of variable. Our compiler leverages this observation to delay stalls for copy statements when possible. This optimization can allow the program to possibly issue more tasks and improve parallelism or even eliminate the stall completely if the variable is overwritten. The transfer function for the copy statement copies the abstract variable sources from y to x if the variable y includes a child task source. Otherwise, the variable's value is immediately available and it operates the same as the transfer function for $\mathrm{wr}(\mathtt{x})$.

**Enter Statement:** When a source from a task flows across a back edge and reaches the enter statement for the same task, some bookkeeping is required because variable sources name a dynamic instance of a task. The enter statement to a task creates a new instance and previous instances of the same task age by one. The transfer function for the statement $\mathtt{enter}(t_{\mathrm{curr}})$ in Figure 2 increments the age of previous variable sources from same task. The analysis becomes finite by bounding the age by $k$; a variable source with $k$ is interpreted as an unknown age. In Section 4.2 we discuss how the compiler generates code to dynamically track such sources.

**Exit Statement:** Recall the discussion in Section 3 of the design decision to simplify the management of task dependence relations by attributing a child task's dependences to its parent task. An implication of this decision is that the transfer function for an exit statement must reflect that a parent task becomes the source for all of its children's values. The transfer function for the $\mathtt{exit}(t_{\mathrm{curr}})$ statement simply identifies variables with at least one child source and changes such variables to have exactly one source from the instance of the current task. The compiler generates code to copy the values from the child task to the current task. Other variable's sources remain unchanged.

#### 4.1.4 Virtual Reads

We introduce virtual reads to simplify the runtime management of variable dependences. Section 3 states that OoOJava handles control dependences between tasks implicitly by forcing them to have a single exit, however a problem remains: which task is the source of a variable when a later task conditionally writes to that variable? We want to maintain the property that OoOJava should not have to search through a series of tasks at runtime to find a variable's value.

Our solution is to treat a conditional write to a variable as a virtual read. The variable dependence analysis identifies this case in the $\mathtt{exit}(t_{\mathrm{curr}})$ transfer function; specifically, it occurs when a live variable at the task exit has a mixture of sources that are from both (1) outside of the current dynamic task instance and (2) from inside the current instance and it children. The transfer function forces such a task to require the variable's value before starting to execute and therefore when the task exits, it has the variable's value.

#### 4.1.5 Termination

Our analysis operates on lattice of finite height and our transfer functions are monotonic. It therefore follows that the variable source analysis terminates.

### 4.2 Code Generation

We next describe how the compiler uses the results of the variable dependence analysis to generate code. We divide code generation for accessing a variable x within task $t_{\mathrm{curr}}$ into three categories based on $M(\mathtt{x})$ at the relevant program point:

**Immediate Access:** When all of the sources in $M(\mathtt{x})$ are from the current task $t_{\mathrm{curr}}$, its ancestors or their siblings, the value is currently stored in the variable. Therefore, normal code is generated to access the variable x immediately.

**Optimized Stall:** When all of the sources in $M(\mathtt{x})$ are from a single instance of a child $t_{\mathrm{child}}$ with an age less than $k$, then the compiler statically knows which dynamic task instance will provide the value for variable x. In this case the generated code should stall the current task until $t_{\mathrm{child}}$ retires and copy the value of x before generating a normal access of x. If the same task is statically known to be the source of other variables, the compiler generates code to also read those values. This optimization avoids the overhead of extra dynamic checks for future accesses to those variables.

**Dynamic Tracking:** Otherwise, the variable dependence analysis cannot track the variable's source statically. In this case, the compiler identifies the points in the control flow graph at which the statically known sources for the variable became unknown. At these points, the compiler inserts code to dynamically track which task generates the variable's value. The compiler must also handle the case in which it is statically unknown whether the variable currently has a value — in this case the code tracks (1) whether the variable has a value, (2) the actual value if it has a value, and (3) the source of value if it does not currently have a value. Then code is generated just prior to the access of x to decide whether the value is available or whether a task instance will provide the value and must be stalled for.

To complete code generation for variable accesses, the compiler also generates some bookkeeping at the exit statement of each task. As mentioned in the task exit transfer function, a parent becomes the source for the values of its children, so the compiler generates code to stall a task until all of its children have retired. Then the task copies the values from the children to itself and makes them available for other tasks to access.

## 5. HEAP DEPENDENCE ANALYSIS

Heap dependences have long posed a challenge to automatically parallelizing code that manipulates data structures. OoOJava reasons about heap accesses in terms of the *heap root* used to reach the accessed heap objects; a heap root is an object referenced by a live variable through which deeper heap references are obtained. Heap roots occur in two contexts: a heap root is either referenced by a variable in the in-set of a taskor is the first object along a heap path accessed by a parent task after the exit of a child task.

Two tasks can only have a heap dependence when both of the following two conditions are true. The first necessary condition is that one task writes to a field $f$ of an object allocated at site $a$ and a second task either reads or writes to $f$ of an object allocated at $a$. We call the pair of accesses potentially conflicting accesses and an object allocated at $a$ a potentially conflicting object. The second necessary condition is that there must exist a potentially conflicting object that is reachable from the heap roots of both tasks that perform the potentially conflicting accesses.

### 5.1 Heap Dependence Overview

We next present a straightforward dynamic approach for preserving heap dependences. We will later extend this basic approach with the static analysis that OoOJava uses to make this approach efficient.

#### 5.1.1 Naive Dynamic Approach

We use a static effects analysis extension to a standard pointer analysis to reports all possible heap accesses a task may execute. Effects are reported as a 4-tuple consisting of: (1) the heap root used to access the affected object, (2) the allocation site (or heap node) of the affected object, (3) the effect (read or write), and finally (4) the affected field.

In our example from Section 2, the effects analysis would report write effects to objects in the AST allocated at several allocation sites due to the call to the `typeCheck` method at Line 7. The analysis would also report read effects to the objects in the AST allocated at several allocation sites due to the calls to both the `typeCheck` method at Line 7 and the `flatten` method at Line 8. In both cases, the affected objects were reached by the `AST` object referenced by the `ast` in-set variable of task `tpar`.

In many cases, the compiler can statically determine that two effects cannot conflict. However, statically checking that the updates to the ASTs do not conflict is difficult because two instances of the task `tpar` may both write to same AST objects. We next describe a dynamic approach that can rule out such conflicts.

To issue an instance of task `tpar` $p$, the system would dynamically check for conflicts between task $p$ and all instances of tasks that have not retired: take, for example, an earlier instance of `tpar`, $p_0$. The runtime traverses the heap reachable from the in-set variable $ast$ to identify all concrete objects that the effects can apply to; the dynamic check would use the allocation site information to prune the set of objects an effect could apply to. If this dynamic check shows that there are no conflicting accesses between the task $p$ and all previous non-retired tasks to the same object instance, then it is safe to immediately invoke $p$. One potential concern is that a data structure may not yet contain all objects that it will when the task is supposed to run. However, in this case there will be a conflict detected on some existing object that will eventually reference the additional objects.

This dynamic approach is not practical because it can incur significant overhead for large data structures. In the next section, we describe how OoOJava uses static analysis to make this basic approach practical.

### *5.1.2 Optimization*

The key insight behind OoOJava is to use reachability results from static analysis combined with simple dynamic checks to make the previous approach practical. For example, if we can determine (1) that all affected objects in an AST are only reachable from at most one AST object through static analysis and (2) that different instances of the task `rblockP` operate on different AST objects through a dynamic check then there is no conflict between the tasks.

We introduce an effects analysis in Section 5.2 to discover a set of effects that conservatively summarize the heap effects of a task. Section 5.3 overviews the approach to reachability analysis that we use to statically compute the reachability of objects from the heap roots in the effects. Section 5.3.2 presents rules for using determining whether two effects conflict. Finally, Section 5.4 describes how OoOJava efficiently implements the dynamic checks to preserve all heap dependences between tasks.

## 5.2 Effects Analysis

OoOJava uses heap effects to conservatively abstract the read and write heap accesses a task may perform.

### *5.2.1 Abstract Domains*

OoOJava represents effects as the 4-tuple $\langle h, a^{\text{aff}}, o, f\rangle \in U \subseteq H \times A \times O \times F$, where $h$ is the heap root used to access the affected object, $a^{\text{aff}} \in A$ is the allocation site of the affected object, $o \in O = \{\texttt{read}, \texttt{write}, \texttt{strong}\}$ is the operation, and $f \in F$ is the affected field.

The analysis uses two different types of heap roots. The first type of heap rootabstracts the objects referenced by a task's in-set variables. The second type abstracts objects referenced by live variables between two task declarations. Formally, a heap root $h$ is given by the tuple $\langle st, v, a\rangle \in H \subseteq ST \times V \times A$, where $st$ is either a stall site or the entrance to a task, $v$ is the in-set or stall site variable, and $a$ is the allocation site of the object referenced by $v$ at location $st$.

The analysis assumes the presence of a pointer analysis. We assume the pointer analysis abstracts objects with a set of heap nodes $n \in N$ and heap references with a set of edges $e \in E \subseteq V \times N \cup N \times F \times N$. We assume that the pointer analysis provides a function $\mathcal{A}$ that maps a heap node to an allocation site. We assume a heap node only abstracts objects allocated at one site — however, modifications to support pointer analysis in which heap nodes abstract objects allocated at multiple sites are straightforward. The analysis computes at each program point the mapping $R \subseteq E \times H$ from edges to the heap rootsthat were used to reach the edge.

| st | $R' = (R - \text{KILL}) \cup \text{GEN}$ |
|---|---|
| `x = ...` | $\text{KILL} = \{\forall\langle e, s\rangle \in R \mid e \in E(\texttt{x})\}$ |
| `x = new` | $\text{GEN} = \emptyset$<br>$\mathcal{L}' = \mathcal{L} \backslash \{\texttt{x}\}$ |
| `x = y` | $\text{GEN} = \{\langle\langle \texttt{x}, n\rangle, s\rangle \mid \forall\langle \texttt{y}, n\rangle \in E, \langle\langle \texttt{y}, n\rangle, s\rangle \in R\}$<br>$\mathcal{L}' = \{v \in V \mid (v \in \mathcal{L} \wedge v \neq \texttt{x}) \vee (\texttt{y} \in \mathcal{L} \wedge v = \texttt{x})\}$ |
| `x = y.f` | $R_a = R \cup \{\langle\langle \texttt{y}, n\rangle, \langle \texttt{st}, \texttt{y}, \mathcal{A}(n)\rangle\rangle \mid \forall\langle \texttt{y}, n\rangle \in E, \texttt{y} \in \mathcal{L}\}$<br>$\text{GEN} = \{\langle\langle \texttt{x}, n'\rangle, s\rangle \mid \forall\langle n, \texttt{f}, n'\rangle \in E(\texttt{y}, \texttt{f}),$<br>$\langle\langle n, \texttt{f}, n'\rangle, s\rangle \in R_a\}$<br>$\mathcal{L}' = \mathcal{L} \backslash \{\texttt{x}, \texttt{y}\}$ |
| `x.f = y` | $R_a = R \cup \{\langle\langle \texttt{x}, n\rangle, \langle \texttt{st}, \texttt{x}, \mathcal{A}(n)\rangle\rangle \mid \forall\langle \texttt{x}, n\rangle \in E, \texttt{x} \in \mathcal{L}\}$<br>$\cup \{\langle\langle \texttt{y}, n\rangle, \langle \texttt{st}, \texttt{y}, \mathcal{A}(n)\rangle\rangle \mid \forall\langle \texttt{y}, n\rangle \in E, \texttt{y} \in \mathcal{L}\}$<br>$\text{KILL} = \emptyset$<br>$\text{GEN} = \{\langle\langle n, \texttt{f}, n'\rangle, s\rangle \mid \forall\langle \texttt{x}, n\rangle \in E, \forall\langle \texttt{y}, n'\rangle \in E,$<br>$\langle\langle \texttt{y}, n\rangle, s\rangle \in R_a\}$<br>$\mathcal{L}' = \mathcal{L} \backslash \{\texttt{x}, \texttt{y}\}$ |
| $\texttt{enter}(t_{\text{curr}})$ | $\text{KILL} = \{\langle e, s\rangle \mid s \text{ is a stall site}\}$<br>$\text{GEN} = \{\langle\langle v, n\rangle, \langle t_{\text{curr}}, v, \mathcal{A}(n)\rangle\rangle \mid v \text{ is an in-set variable for } t_{\text{curr}} \wedge$<br>$\langle v, n\rangle \in E\}$<br>$\mathcal{L}' = \emptyset$ |
| $\texttt{exit}(t_{\text{curr}})$ | $\text{KILL} = \{\langle e, s\rangle \mid s \text{ is a stall site} \vee s \text{ is an in-set variable for the}$<br>$\text{current instance of } t_{\text{curr}}\}$<br>$\text{GEN} = \emptyset$<br>$\mathcal{L}' = V$ |

**Figure 3: Transfer Functions for Computing Heap Roots**

The analysis computes at each program point a set of variables $\mathcal{L}$ that the application may have to stall before reading if the variable references a data structure for which there is a conflict.

The mappings $R$ and $\mathcal{L}$ both form lattices. The partial order ($\sqsubseteq$) is defined by the subset relation ($\subseteq$); join($\sqcup$) is set union ($\cup$); bottom($\perp$) is the empty set($\emptyset$); and top($\top$) is the maximally full set. The lattices are of finite height because their domains are finite.

The analysis generates a set of effects $U$ for the program. Note that there is only one set of effects for the entire program.

### *5.2.2 Transfer Functions*

We decompose the effects analysis into two passes. The first pass computes the mapping $R$ from edges to the heap roots used to access objects. The second pass then uses the mapping $R$ to compute the application's set of effects $U$.

Figure 3 presents the transfer functions for computing the mapping $R$ from reference edges in the points-to graph to heap roots. The analysis introduces new heap roots into points-to graphs at two classes of statements: (1) task enter statements and (2) statements of a parent task that may have to stall to avoid heap conflicts with a child task. These statements create new heap rootsfor the corresponding variable's edges. Heap roots are then subsequently propagated to newly-created references because we are interested in abstracting which heap root was used to access an affected object. **[[talk more about heap roots here]]**

The heap roots analysis uses a simple supporting analysis that pre-computes which variables may require stalls. This supporting analysis is relevant for sections of a task following the exit of a child task. The goal of this analysis is to compute the set of variables $\mathcal{L}$ for which reads from these variables may require the generation of a stall. Figure 3 presents the transfer functions for computing the set $\mathcal{L}$. At the exit of a child task, the set $\mathcal{L}$ contains all live variables that reference objects. At the entrance of a task, the set $\mathcal{L}$ is empty because all data structures can be accessed without needing to stall

| st | $U' = U \cup GEN$ |
|---|---|
| `x=y.f` | GEN $= \{\langle s, \mathcal{A}(n), \mathtt{read}, \mathtt{f}\rangle \mid \forall \langle \mathtt{y}, n\rangle \in E,$ $\langle\langle \mathtt{y}, n\rangle, s\rangle \in R\}$ |
| weak `x.f=y` | GEN $= \{\langle s, \mathcal{A}(n), \mathtt{read}, \mathtt{f}\rangle \mid \forall \langle \mathtt{x}, n\rangle \in E,$ $\langle\langle \mathtt{x}, n\rangle, s\rangle \in R\}$ |
| strong `x.f=y` | GEN $= \{\langle s, \mathcal{A}(n), \mathtt{read}, \mathtt{f}\rangle \mid \forall \langle \mathtt{x}, n\rangle \in E,$ $\langle\langle \mathtt{x}, n\rangle, s\rangle \in R\} \cup \{\langle s, n, \mathtt{strong}, *\rangle \mid$ $\forall \langle \mathtt{x}, n_0\rangle \in E, \forall n \in N, \langle\langle \mathtt{x}, n_0\rangle, s\rangle \in R \wedge$ $n$'s reachability decreases$\}$ |

**Figure 4: Transfer functions for effects.**

for child tasks. The transfer functions for $\mathcal{L}$ remove a variable at a statement that reads the variable and therefore serves as a potential stall site for the data structure referenced by the variable.

Figure 4 presents the transfer functions for computing heap effects. Load statements and store statements are the only statements that operate on the object fields in the heap and therefore generate effects. These transfer functions record for each field access: the heap rootthat was used to reach the object, the allocation site of the object, the operation, and the field the statement accessed. The analysis simply accumulates effects into a global set $u$. The effects analysis treats array operations as normal field accesses on a special array field. **[[What to do about strong updates...kind of early to address them here...]]**

### 5.2.3 *Interprocedural Extension*

We next present the interprocedural extension to the effects analysis. to return interprocedurally-relevant information. The points-to analysis we used handles method calls by taking a snapshot of the caller's heap just prior to the method invocation and labeling the heap elements with predicates—these are tautologies that are true for the given caller heap. For example, the predicate for an edge $\langle n_1, f, n_2\rangle$ is that the edge $\langle n_1, f, n_2\rangle$ existed in the caller. Then the heap contexts for all of a method callers are merged. The predicates serve to preserve analysis precision by preventing the erroneous propagation of edges from one caller to another. The interprocedural extension also adds predicates to heap roots: heap root predicates specify that a given edge in the caller had the given heap root. There is a special predicate for heap roots for the current method context — the analysis uses this predicate to determine which heap roots to prune at task exits (in case of recursive tasks).

### 5.2.4 *Basic Conflict Detection*

OoOJava considers a task to have a heap dependence on an earlier task when an effect of one task conflicts with an effect of the other. Conflicts due to the effect elements are simply determined. Consider an task $t_0$ with the effect $\langle h_0, a_0^{\text{aff}}, o_0, f_0\rangle$ and an task $t_1$ with the effect $\langle h_1, a_1^{\text{aff}}, o_1, f_1\rangle$:

1. If $a_0^{\text{aff}} \neq a_1^{\text{aff}}$, then there is no conflict because the objects must be different if they were allocated at different sites.
2. If $o_0 = o_1 = \mathtt{read}$, then there is no conflict because reads do not conflict.
3. If $f_0 \neq f_1$, then there is no conflict because the two effects access different fields.

When these rules cannot eliminate a conflict, OoOJava will consider reachability to possibly eliminate the conflict or generate a simple dynamic check for the conflict.

## 5.3 Reachability-based Conflict Detection

Disjoint reachability analysis is a new static analysis that conservatively extracts reachability properties between heap objects. OoOJava uses these reachability properties to generate lightweight dynamic checks that can rule out conflicts. When OoOJava cannot eliminate a conflict between two effects using the basic conflict detection rules, it uses disjoint reachability analysis to compute the reachability from the heap nodes that correspond to the allocation sites in the effects' heap roots. Section 5.3.1 presents an overview of disjoint reachability analysis. Section 5.3.2 presents the reachability-based conflict detection rules that OoOJava uses to eliminate the possibility of conflicts between effects when the basic conflict resolution rules cannot.

### 5.3.1 *Disjoint Reachability Analysis*

Disjoint reachability analysis extends a standard pointer analysis with reachability states. An object's reachability state conservatively characterizes the set of objects that can possibly reach the given object. Disjoint reachability analysis is demand driven — it only computes reachability from objects of interest to the compiler. OoOJava uses the results from the effects analysis to generate a set of allocation sites of interest.

A reachability state is a set of heap node-arity pairs with the constraint that no two pairs in a reachability state have the same heap node. The arity element of the pair is taken from the set $\{\mathtt{0}, \mathtt{1}, \mathtt{MANY}\}$. A reachability state that abstracts the reachability of an object $o$ that includes the pair $\langle n_1, \mu\rangle$ means that at most $\mu$ objects abstracted by heap node $n_1$ have paths through the heap to object $o$. The arity `0` means exactly zero, `1` means at most one, and `MANY` means any number of objects. We omit reachability tuples with arity `0`.

The analysis associates a set of reachability states $S$ with each heap node $n$ in the points-to graph. For each object $o$ that is abstracted by heap node $n$ there is a reachability state $s \in S$ that conservatively abstracts the reachability of $o$.

The analysis also associates a set of reachability states $S$ with each edge $e \in E$ in the points-to-graph. Reachability states associates with an edge characterize the reachability of the objects reachable from that edge. Intuitively, these reachability states serve to provide a precise way to propagate reachability changes through the heap. For each object $o$ and for each sequence of references $r_1; r_2; ...; r_j$ in the heap that form a path to $o$, there is a reachability state $s$ that conservatively abstracts the reachability of object $o$ in the set of reachability states for each edge $e_1; e_2; ...; e_j$, where edge $e_i$ abstracts reference $r_i$. The details of disjoint reachability analysis including formalization are available in a technical report [3].

```
1  x = new SharedObj(); // objs abstracted by n1
2  y = new Foo();       // objs abstracted by n2
3  z = new Bar();       // objs abstracted by n3
4  if( ... ) {
5    y.f = x;           // obj in n2 reaches obj in n1
6  } else {
7    z.b = x;           // obj in n3 reaches obj in n1
8  }
```

**Figure 5: Code fragment for reachability example.**

Figure 5 presents an example that we will use to illustrate disjoint reachability analysis. At the exit of Line 8 the set of reachability states for heap node $n_1$, which abstracts the shared objects in this short example, only includes $[\langle n_2, \mathtt{1}\rangle]$ and $[\langle n_3, \mathtt{1}\rangle]$. This set of reachability state shows that a shared object from allocation site 1 cannot be reachable from both an object from site 2 and site 3 at this program point. If that were a possibility (by moving Line 7 to just after Line 5, for instance), then the reachability state $[\langle n_2, \mathtt{1}\rangle, \langle n_3, \mathtt{1}\rangle]$ would be present in the set of reachability states associated with $n_1$ at the exit of Line 8.

### 5.3.2 *Reachability-based Effect Conflict Elimination*

In the running example presented in Section 2 we noted that a necessary condition for safely executing instances of task `tpar` in parallel is to ensure two given instances do not have heap dependences, specifically with respect to objects in an AST. Assume `AST` objects are allocated at the same site and are summarized in the points-to graphs by heap node $n_{\text{AST}}$. Next assume that an `AST` object serves as a data structure header object for a collection of `TreeNode` objects summarized by $n_{\text{TN}}$. Disjoint reachability information associated with $n_{\text{TN}}$ is critical to OoOJava for deciding when to safely execute instances of task `tpar`. If the set of reachability states associated with $n_{\text{TN}}$ includes at state with reachability tuple $\langle n_{\text{AST}}, \mathtt{MANY}\rangle$ then a given `TreeNode` object may be reachable from any number of `AST` header objects, and therefore we cannot be certain whether any two instances of `tpar` might access a common `TreeNode` object. However, in the running example we find only $\langle n_{\text{AST}}, \mathtt{1}\rangle$, meaning a given `TreeNode` object is reachable from at most one `AST`. When a dynamic check determines two instances of `tpar` have references to distinct `AST` objects then we can be sure they access disjoint sets of `TreeNode` objects, and therefore are safe to execute concurrently.

If the basic conflict detection rules cannot eliminate a possible conflict between two effects, the results of disjoint reachability analysis may rule out a conflict by stating whether the effects of the tasks in question operate on disjoint sets of objects or not.

Consider the two extended effects: $\langle\langle st_0, v_0, a_0\rangle, a^{\text{aff}}, o_0, f\rangle$ and $\langle\langle st_1, v_1, a_1\rangle, a^{\text{aff}}, o_1, f\rangle$.

OoOJava inspects the set of reachability states $S$ for each heap node associated with $a^{\text{aff}}$ to decide whether the effects conflict:

**1.** If $a_0^{\text{root}} \neq a_1^{\text{root}} \wedge \forall s \in S.s \cap (\{\langle a_0, \mathtt{1}\rangle, \langle a_0, \mathtt{MANY}\rangle\} \times \{\langle a_1, \mathtt{1}\rangle, \langle a_1, \mathtt{MANY}\rangle\}) \neq \emptyset$, then there is not conflict. If there is no such reachability state then the two heap roots cannot access the same object allocated at allocation site $a^{\text{aff}}$ and therefore the effects do not conflict.

**2.** If $a_0 = a_1 \wedge \forall s \in S.s \cap \langle a_0, \mathtt{MANY}\rangle = \emptyset$, then the compiler can generate a simple dynamic check to eliminate the possibility of a conflict. If $a_0^{\text{root}} \neq a_1^{\text{root}}$ at runtime, then there is no conflict. We call conflicts that can be eliminated by a dynamic check fine-grained conflicts.

**3.** If neither the basic nor reachability-based conflict detection can eliminate a conflict, the effects are said to have a coarse-grained conflict.

It is important to note that reachability graphs are not generally comparable between different program points. We make the observation that if there is no strong update, and therefore no decrease in reachability information, on any control path from one program point to another, then it is safe to compare reachability graphs. Extended effects analysis adds `strong` to the domain of possible effects and generates a strong update effect whenever a write effect decreases the reachability graph obtained at that same program point. Effects of type `strong` always conflict with other effects on the same affected object allocation site and result in a coarse-grained conflict.

## 5.4 Code Generation

OoOJava assembles all effect conflicts between tasks (including an instance of a task with previous instances of itself) into a heap effect conflict graph, discussed in Section 5.4.1. A series of heap dependence queues implements the conflict graph efficiently at runtime; Section 5.4.2 discusses the queue properties and Section **??** gives the algorithm for finding the set of heap dependence queues that covers the conflict graph. In Section 5.4.3 we give the actual process for generating code to issue a task by inserting it into relevant heap dependence queues.

### *5.4.1 Conflict Graph*

OoOJava generates conflict graphs from the results of the effects analysis and the disjoint reachability analysis. There is a conflict graph for each task that summarizes the heap dependences between that taskand its child tasks. There is a node in a conflict graph for each heap root. There are two types of edges in the conflict graph: coarse-grained edges indicate that the corresponding code blocks cannot be reordered and fine-grained edges indicate the code blocks can only be reordered if the corresponding heap roots refer to different objects. Without a loss of generality, whenever there is both a fine-grained and coarse-grained edge between the same two nodes, we drop the fine-grained edge because the ordering constraints of the coarse-grained edge dominate.

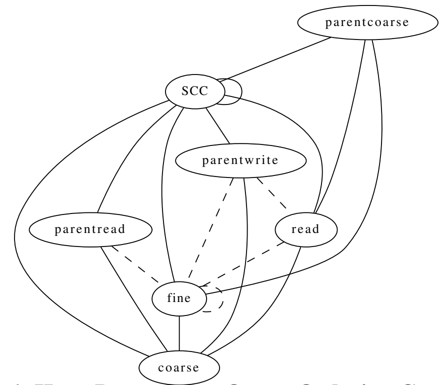


**Figure 6: Heap Dependence Queue Ordering Constraints**

### *5.4.2 Heap Dependence Queues*

OoOJava compiles each conflict graph into a set heap dependence queues that enforce the conflict graph's ordering constraints. Each heap dependence queue accepts the following types of entries: read, write-fine, write-coarse, parent-read, parent-write-fine, parent-write-coarse, and self-write-coarse. Figure 6 illustrates the ordering constraints that a single heap dependence queue enforces between entries of different types. A dashed edge indicates a fine-grained order constraint that heap roots of the corresponding types can be only reordered if the refer to different objects. A solid edge indicates a coarse-grained order constraint that heap roots of the corresponding types can never be reordered. Section **??** further discuss how we efficiently implement the heap dependence queues.

We formulate compiling a conflict graph into a set of heap dependence queues as a covering graph problem. Each edge in the conflict graph must be enforced by an edge in heap dependence queue's constraint graph. The compiler maps nodes in the conflict graph to nodes in the heap dependence queues. An edge in the conflict graph is enforced by the heap dependence queue if the edge's endpoints in the conflict graph map to nodes in a heap dependence queue with the same type of edge between them.

The graph covering problem is a generic graph covering problem; Figure 7 presents a greedy algorithm for heuristically solving it with a minimal set of heap dependence queues. Line 2 creates a new heap dependence queue. Lines Lines 3-10 greedily cover the fine-grained edges in the conflict graph that the heap dependence queue will enforce, and Lines 11-22 greedily cover the coarse-grained edges. The algorithm greedily covers edges in the conflict graph until the the given heap dependence queue cannot cover any more edges; the process repeats until every conflict edge in the conflict graph has been covered.

**[[fix from here forward]]** Line 5 tests whether an uncovered node $u$ incident to an uncovered fine-grained edge $(u, w)$ might be covered by the queue; it will be covered by the queue if it has a self edge (which implies the node requires write access) and there is an edge $(u, x)$ to every other node $x$ that is already covered, meaning

Cover(conflict graph $C = (V, V_p, E, E_f, E_c)$)
1. set of covered edges $E' \leftarrow \emptyset$
2. create new queue $q \leftarrow (r = \emptyset, wf = \emptyset, wc = \emptyset, pr = \emptyset,$
$pwf = \emptyset, pwc = \emptyset, swc = \emptyset)$
3. for each $(u, w) \in E_f$
4. if $(u, w) \in E' \vee$QCovFine$(q, u)$ then goto 3
5. if $(u, u) \in E \wedge \forall x.$ QCovFine$(q, x) \implies (u, x) \in E$ then
6. $E' \leftarrow E' \cup \{(u, v) \mid \forall v \in V.$ QCovFine$(q, v)\}$
7. if $u \in V_p$ then $pwf \leftarrow pwf \cup \{u\}$ else $wf \leftarrow wf \cup \{u\}$
8. else if $(u, u) \notin E \wedge$
$\forall x.$ QCovFnWr$(q, x) \implies (u, x) \in E$ then
9. $E' \leftarrow E' \cup \{(u, v) \mid \forall v \in V.$ QCovFnWr$(q, v)\}$
10. if $u \in V_p$ then $pr \leftarrow pr \cup \{u\}$ else $r \leftarrow r \cup \{u\}$
11. for each $(u, w) \in E_c$
12. if $(u, w) \in E' \vee$QCov$(q, u) \vee$PrntCoar$(q, (u, w))$ then
13. goto 11
14. if $u \in V_p \wedge \forall x.$QCovFnCh$(q, x) \implies (u, x) \in E_c$ then
15. $E' \leftarrow E' \cup \{(u, v) \mid \forall v \in V.$ QCovFnCh$(q, v)\}$
16. $pwc \leftarrow pwc \cup \{u\}$
17. else if $u \notin V_p \wedge (u, u) \notin E \wedge$
$\forall x.$ QCovFine$(q, x) \implies (u, x) \in E_c$ then
18. $E' \leftarrow E' \cup \{(u, v) \mid \forall v \in V.$ QCovFine$(q, v)\}$
19. $wc \leftarrow wc \cup \{u\}$
20. else if $u \notin V_p \wedge (u, u) \in E_c \wedge \forall x.$ QCov$(q, x) \implies (u, x) \in E_c$ then
21. $E' \leftarrow E' \cup \{(u, v) \mid \forall v \in V.$ QCov$(q, v)\} \cup \{(u, u)\}$
22. $swc \leftarrow swc \cup \{u\}$
23. $Q \leftarrow Q \cup \{q\}$
24. if $\exists e \in E. e \notin E'$ then goto 2 else return $Q$

QCov($(r, wf, wc, pr, pwf, pwc, swc)$, $u$)=
$(u \in r \vee u \in wf \vee u \in wc \vee u \in pr \vee u \in pwf \vee$
$u \in pwc \vee u \in swc)$

QCovFine($(r, wf, wc, pr, pwf, pwc, swc)$, $u$)=
$(u \in r \vee u \in wf \vee u \in pr \vee u \in pwf)$

QCovFnWr($(r, wf, wc, pr, pwf, pwc, swc)$, $u$)=
$(u \in wf \vee u \in pwf)$

QCovFnCh($(r, wf, wc, pr, pwf, pwc, swc)$, $u$)=
$(u \notin V_p \wedge (u \in r \vee u \in wf))$

PrntCoar($(r, wf, wc, pr, pwf, pwc, swc)$, $(u, w)$)=
$(u \notin V_p \wedge v \in pwc \wedge (u, u) \notin E_c)$

**Figure 7: Conflict graph covering algorithm.**

$(u, w)$ grows a fine-covered clique. If so, all the fine-grained edges from $u$ to already covered nodes are considered covered, and $u$ is added to the queue's parent-write-fine or write-fine depending on whether $u$ is a stall site node or not. The set $u$ is added to identifies which waiting type an instance of the task related to $u$ should be inserted into the queue as. Similar to Lines 5-8, Lines 9-12 may cover a new node $u$ and some fine-grained edges incident to $u$, except possibly covering $u$ as the parent-read or read waiting element types. The condition that there is no self-edge $(u, u)$ implies $u$ requires only read access.

Lines 13-24 consider covering an uncovered node $u$ incident to an uncovered coarse-grained edge $(u, w)$. Line 14 tests that $(u, w)$ is uncovered and also a special condition so that when either are true the algorithm should continue to the next coarse-grained edge. The special condition, tested by predicate PrntCoar(, is whether the given edge $(u, w)$ is a non-parent node $u$ incident to a node $w$ already covered as the parent-write-coarse waiting element type and that $u$ has no self-edge. If this special condition is true the edge $(u, w)$ should not be covered by this queue because, as Figure 6 demonstrates, heap dependence queues by design do not enforce this ordering constraint between waiting element types. The edge $(u, w)$ will simply be covered by another queue. Otherwise, $(u, w)$ may be covered with $u$ in any of three coarse-grained waiting element types.

Uncovered node $u$ will be covered as parent-write-coarse if it is a parent stall site node and it has a coarse-grained edge to every read and write-fine node already covered by the queue. If $u$ is not a parent stall site node and there is no self edge $(u, u)$, we consider covering it as write-coarse if it has a coarse-grained edge with every node already covered by the fine-grained types. The last case is to possibly cover $u$ as the self-write-coarse when it has a coarse-grained edge to itself and every other node already covered by the queue; we cover nodes as self-write-coarse specially because these nodes can never be reordered in the heap dependence queue under any circumstance, unlike the other waiting element types.

### 5.4.3 Implementation

After the conflict graph is compiled into heap dependence queues, OoOJava is able to generate code to insert tasks into heap dependence queues to fully respect all heap dependences.

OoOJava issues a task by inserting it into all relevant queues as a waiting element of type determined by the graph-covering solution. Similarly, at a program point identified as a parent stall site, OoOJava inserts the currently executing parent task into the relevant queue and of the calculated waiting element type. A task may not execute while it is not at the head of any heap dependence queue.

The implementation of the heap dependence queue manages which subsequences of tasks may be executed out-of-order, dictated by the waiting element types of the tasks. Heap dependence queues are designed to provide constant-time operations to the OoOJava runtime which lead to the set of possible types. Any consecutive sequence of fine-grained waiting elements are grouped in a hashtable within a single element in the heap dependence queue. The hash keys are unique object IDs that persist across garbage collection, and in the case of collisions, tasks are placed on a list for that key which preserves their required execution order. Hash collisions cause false dependences but are rare. In this way OoOJava can respect fine-grained heap dependences with constant time hashtable operations.

Any consecutive sequence of coarse-grained waiting elements inserted into the heap dependence queue are grouped into one list element. No fine-grained conflicts inserted into the heap dependence queue before or after coarse-grained elements may be reordered across them. A special property of both the fine-grained hashtable elements and coarse-grained list elements in the heap dependence queue is that when only parent elements are left in the subelements, the entire subelement is removed from the heap dependence queue. The observation is that the various statements of the parent that might conflict with children cannot conflict with each other, as the serial execution of a given parent task already guarantees an order for its heap operations.

The self-write-coarse element is a special case in that it may never be reordered with any other waiting queue types including its own type. Therefore, each task inserted with type self-write-coarse is a single element in the heap dependence queue and nothing may be reordered across it.

OoOJava will only see a performance improvement if the dynamic overhead of the generated code to respect heap dependences allows sufficient out-of-order execution. In our experience the static analysis that supports fine-grained heap dependences with constant time dynamic checks is worth the cost, statically and dynamically.

## 6. DISCUSSION

We present one strict and one relaxed strategy that guarantee a OoOJava implementation's I/O behavior matches its serial elision. The most strict and correct strategy is to protect all access to file descriptors as members of a singleton I/O object. In this strategy every I/O operation is serialized and the behavior will match the serial elision in every case.

The strict strategy prohibits parallelism when unrelated tasks access disjoint sets of file descriptors. We can relax the strategy by assuming that file descriptors are never aliased and then safely serial access to each file descriptor separately. We expect the relaxed strategy is sufficient for the correct behavior of many programs, and has potential for significantly more parallelism.

| Benchmark | Measured Speedup | Lines |
|---|---|---|
| RayTracer | 23.14× | 3,385 |
| KMeans | 14.99× | 3,553 |
| Labyrinth | 14.79× | 4,644 |
| Barnes-Hut | 14.76× | 3,479 |
| Power | 8.58× | 2,282 |
| Crypt | 9.94× | 2,471 |
| MonteCarlo | | |
| MergeSort | 18.6× | 2,331 |

**Figure 8: Speedups on Benchmarks (Higher is Better)**

OoOJava takes as input a subset of Java which excludes threads, reflection, and exceptions. Explicit threads are purposefully excluded in the sense that the task annotation provided by OoOJava is an alternative to explicit threads. Supporting reflection would require more complex abstract domains and most likely have an adverse effect on both the performance and accuracy of the static analyses.

Exceptions could be handled precisely in OoOJava when an exception does not escape a task. Otherwise, exceptions can still be supported but with imprecise semantics.

## 7. EVALUATION

We have implemented OoOJava and evaluated it on a 1.9 GHz 24-core AMD Magny-Cour Opteron with 16 GB of memory. Our implementation and benchmarks are available on the web. We report times averaged over 10 runs. Figure 8 summarizes the measured speedups and lines of code including libraries for each of our benchmarks.

**RayTracer:** RayTracer was taken from the Java Grande Benchmark suite [24].
**KMeans:** KMeans was taken from the STAMP benchmark suite [9].
**Labyrinth:** Labyrinth was taken from the STAMP benchmark suite [9].
**Barnes-Hut:** Barnes-Hut was taken from the Lonestar benchmark suite [15].
**Power:** Power was taken from the JOlden [8] benchmark suite.
**Crypt:** MonteCarlo was taken from the Java Grande Benchmark suite [24].
**MonteCarlo:** MonteCarlo was taken from the Java Grande Benchmark suite [24].
**MergeSort:**

**[[A nice piece to add to the discussion here is whether a benchmark had fine, coarse or any heap conflicts and a brief explanation.]]**

## 8. RELATED WORK

This work presents system details and an evaluated implementation of OoOJava as previously described [1].

Some earlier deterministic parallel programming models focus on specific code segments, such as loops [11]. More recently, Grace [5] offers a deterministic model for the common fork-join parallel pattern.

OoOJava and other models target more general code. One approach is to guarantee deterministic execution under unchecked conditions [25, 12, 19], such assuming disjoint data structures. Another approach is to guarantee behavior through compiler analysis; CellSs [4] and systems evaluated by Loidl [16] simplify the required analysis by constraining the use of data structures. A third alternative is to allow general data structures but require developer annotations to characterize accesses [21, 7, 2]. OoOJava differs from these works by targeting general code and using a single, simple annotation to provide a deterministic model, lifting the correct determination of dependences off of the developer through new static and dynamic techniques. Generally, OoOJava attempts to deliver a competitive parallel implementation for minimal effort over the sequential development.

DSWP [18] also uses static analysis to determine dependences, specifically Cheng and Hwu's pointer analysis [10]. However, disjoint reachability analysis leveraged by OoOJava can capture reachability properties that previous pointer analysis and shape analysis [22, 17] cannot.

Deterministic, speculative models like BOP [13] offer simple annotations similar to OoOJava but must incur overhead to support general protection.

OoOJava differs from inspector-executor approaches [20, 23] in that it supports complex object-oriented data structures, uses the results of static analysis to avoid inspecting nearly all memory accesses, and does not require a runtime preprocessing phase.

# 9. CONCLUSION

We believe there is a common class of programs for which this technique provides a competitive parallel implementation for comparably low cost.

# 10. REFERENCES


[1] Anonymized workshop paper.

[2] M. D. Allen, S. Sridharan et al. Serialization sets: a dynamic dependence-based parallel execution model. In *PPoPP*, 2009.

[3] Anonymized Technical Report. Disjoint reachability analysis. Tech. rep., 2010.

[4] P. Bellens, J. Perez et al. CellSs: a programming model for the Cell BE architecture. *SC*, 2006.

[5] E. D. Berger, T. Yang et al. Grace: safe multithreaded programming for C/C++. In *OOPSLA*, 2009.

[6] R. L. Bocchino, Jr., V. S. Adve et al. Parallel programming must be deterministic by default. In *HotPar*, 2009.

[7] R. L. Bocchino, Jr., V. S. Adve et al. A type and effect system for deterministic parallel Java. In *OOPSLA*, 2009.

[8] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *PACT*, 2001.

[9] C. Cao Minh, J. Chung et al. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

[10] B.-C. Cheng and W.-m. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI*, 2000.

[11] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 1998.

[12] J. S. Danaher, I.-T. A. Lee et al. The JCilk language for multithreaded computing. In *SCOOL*, 2005.

[13] C. Ding, X. Shen et al. Software behavior oriented parallelization. In *PLDI*, 2007.

[14] P. Jouvelout and D. Gifford. The FX-87 interpreter. In *ICCL*, 1988.

[15] M. Kulkarni, M. Burtscher et al. Lonestar: A suite of parallel irregular programs. In *ISPASS*, 2009.

[16] H.-W. Loidl, F. Rubio et al. Comparing parallel functional languages: Programming and performance. *HOSC*, 2003.

[17] M. Marron, M. Méndez-Lojo et al. Sharing analysis of arrays, collections, and recursive structures. In *PASTE*, 2008.

[18] G. Ottoni, R. Rangan et al. Automatic thread extraction with decoupled software pipelining. *MICRO*, 2005.

[19] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. Massachusetts Institute of Technology, 1998.

[20] L. Rauchwerger, N. M. Amato et al. Run-time methods for parallelizing partially parallel loops. In *ICS*, 1995.

[21] M. C. Rinard, D. J. Scales et al. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 1993.

[22] M. Sagiv, T. Reps et al. Parametric shape analysis via 3-valued logic. In *TOPLAS*, 2002.

[23] J. H. Saltz and R. Mirchandaney. Run-time parallelization and scheduling of loops. *TC*, 1991.

[24] L. A. Smith, J. M. Bull et al. A parallel Java Grande benchmark suite. In *SC*, 2001.

[25] J. Subhlok and B. Yang. A new model for integrated nested task and data parallel programming. In *PPoPP*, 1997.

[26] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. of Res. Dev.*, 1967.