OoOJava: Software Out-of-Order Execution

James C. Jenista Yong hun Eom Brian Demsky

Abstract

Developing parallel software using current tools can be challenging. Even experts find it difficult to reason about the use of locks and often accidentally introduce race conditions and deadlocks into parallel software. We present OoOJava, a new approach to parallel programming inspired by out-of-order processors. Out-of-order processors have long extracted unstructured parallelism from sequential instruction streams. In our approach, a developer annotates code blocks as tasks to decouple these blocks from the parent execution thread. OoOJava extracts 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 an average speedup of 15.3× on our nine 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 currently ship with as many as twelve cores and processors with as many as 1,000 cores will become commonplace within a few years [7]. Software development tools lag behind hardware platforms; developing parallel software using current tools is both difficult and error-prone. They 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 applications 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) [10], constrains the usage of data structures [17, 4], requires extensive annotations [22, 6], or only guarantees determinism under unchecked conditions (disjoint data structures) [25, 12, 20].

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 [15]. OoOJava uses the results of 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 for variables. Together, these techniques allow OoOJava to parallelize a wide-range of programs while requiring few changes to sequential code.

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's similarity 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 evaluated its performance on nine 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 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 it can decouple the

annotated block of code from the parent thread's execution. 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. Task must have a single exit.

In our example, we enclosed Lines 7 and 8 in the task `tpar` to decouple their execution from the main thread. An instance of the task `tpar` can be safely executed in parallel if the variable `ast` references a different object than it does in previous iterations, and the objects in the AST that are updated are only reachable from one `AST` object. Parallelizing this loop poses a problem for most automatic parallelization systems — it is difficult to statically determine that different loop iterations access different `AST` objects.

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 a 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 the serial elision. Annotations in OoOJava never affect the program's correctness, but merely its performance.

3. Execution Model

OoOJava's execution model 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's 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 skips to the end of the task declaration to immediately resume executing its own code. OoOJava dispatches a task when its dependences are fully resolved. When a 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. Tasks in OoOJava form a tree at runtime. A parent is responsible for issuing its children and managing the dependences among its children and itself.

A key 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. OoOJava handles control dependences implicitly by constraining a task to have a single exit. One implication is that a task should not throw an exception that it does not catch. Section 6 discusses the constraints OoOJava places on exceptions in more detail.

```
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

A task may have a data dependence on another task through a variable or through conflicting heap accesses to the same object. Different runtime instances of the same task reference the same variables. Like register renaming in out-of-order hardware, a critical 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 parent-child or sibling-sibling in nature. OoOJava enforces this structure by attributing a child's dependences to its parent and only retiring a task after all of its 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 last 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 the program's execution propagates values in variables between tasks.

To be more precise, the variable source for the variable v_1 has the form $\langle t, g, v_2 \rangle \in S \subseteq T \times G \times V$. The combination of a task $t \in T$ and an age $g \in G$ of that task (i.e. how many instances of that task have been issued between the source instance and the current program point) together statically specify a dynamic instance of a task. To bound the analysis, g is taken from the set $G = \{0, 1, \ldots, k\}$, where a variable source with $g = k$ means an instance with an unknown age and $g = 0$ means the most recent instance. The variable $v_2 \in V$ specifies which variable at the exit of the task given by t and g contains the relevant value. A variable source provides a complete abstract address for the variable's value.

Each task has a set of in-set variables and a set of out-set variables. A task's in-set variables are variables read by the task that were written to before it began executing. Similarly, out-set variables are variables a task writes 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 {`ast`} and the out-set {`cfg`}

while task `tser` has the in-set $\{\texttt{m2cfg}, \texttt{m}, \texttt{cfg}\}$. Task in-sets and out-sets combined with variable source tokens provide the necessary information to route the values of variables between tasks.

4.1.1 Abstract Domains

The analysis is structured as a standard forward dataflow analysis. There is a set L for each program point of the live variables at the entrance to that program point. 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 possible sources 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($\bot$) is the empty set($\emptyset$); and top($\top$) is the maximally full set. The lattice M has a 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, the analysis views method invocations as a read of every argument followed by 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 $\texttt{exit}(t)$.

4.1.3 Transfer Functions

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

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 $\langle \texttt{x}, \langle t, 0, \texttt{x}\rangle\rangle$, 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 optimizations to reduce the overhead of communicating values through variables from one task 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 stall (wait) for the child task to complete and then 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 rule 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 reference currently available values.

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 variables reference currently available values.

st	$M' = (M - \text{KILL}) \cup \text{GEN}$
$\mathrm{wr}(\texttt{x})$	$\text{KILL} = \{\texttt{x}\} \times M(\texttt{x})$ $\text{GEN} = \{\langle \texttt{x}, \langle t_{\text{curr}}, 0, \texttt{x}\rangle\rangle\}$
$\mathrm{rd}(\texttt{x})$	Single child sources: If $M(\texttt{x}) = \{\langle t_{\text{child}}, g, v_1\rangle, \ldots, \langle t_{\text{child}}, g, v_m\rangle\} \wedge$ t_{child} is a child of t_{curr}, $Y = \{\forall y \in L \mid \exists w_1, \ldots, w_k,$ $M(y) = \{\langle t_{\text{child}}, g, w_1\rangle, \ldots,$ $\langle t_{\text{child}}, g, w_k\rangle\}\}$ $\text{KILL} = \{\langle y, s\rangle \mid \forall y \in Y, \forall s \in M(y)\}$ $\text{GEN} = \{\langle y, \langle t_{\text{curr}}, 0, y\rangle\rangle\} \mid \forall y \in Y\}.$
	Mixed sources case: If $M(\texttt{x}) = \{\langle t_{\text{child}}, g_1, v_1\rangle, \langle t_2, g_2, v_2\rangle, \ldots\} \wedge$ t_{child} is a child of t_{curr}, $\text{KILL} = \{\texttt{x}\} \times M(\texttt{x}),$ $\text{GEN} = \{\texttt{x}\} \times \{\langle t_{\text{curr}}, 0, \texttt{x}\rangle\}.$
	No child sources case: $\text{KILL} = \text{GEN} = \emptyset$
$\texttt{x} = \texttt{y}$	$\text{KILL} = \{\texttt{x}\} \times M(\texttt{x})$ $\text{GEN} = \{\langle t, g, v\rangle \in M(\texttt{x}) \mid t \text{ is a child } t_{\text{curr}}\} \cup$ $\{\langle t_{\text{curr}}, 0, \texttt{x}\rangle \mid \langle t, g, v\rangle \in M(\texttt{x}) \wedge$ $\neg t \text{ is a child } t_{\text{curr}}\}$
$\texttt{enter}(t_{\text{curr}})$	$\text{KILL} = \{\langle t, g, w\rangle \mid \forall \langle t, g, w\rangle \in M \wedge t = t_{\text{curr}}\}$ $\text{GEN} = \{\langle t, g \oplus 1, w\rangle \mid \forall \langle t, g, w\rangle \in M \wedge t = t_{\text{curr}}\}$ $g_1 \oplus g_2 = \begin{cases} g_1 + g_2 & \text{if } g_1 + g_2 < k, \\ k & \text{otherwise.} \end{cases}$
$\texttt{exit}(t_{\text{curr}})$	$Z = \{z \in L \mid M(z) = \{\langle t, g, v\rangle, \ldots\} \wedge$ $(t \text{ is a child of } t_{\text{curr}} \vee t = t_{\text{curr}})\}$ $\text{KILL} = \{\langle z, s\rangle \mid \forall z \in Z, \forall s \in M(z)\}$ $\text{GEN} = \{\langle z, \langle t_{\text{curr}}, 0, z\rangle\rangle \mid z \in Z\}.$

Figure 2. Transfer Functions for Variable Dependence Analysis (t_{curr} denotes the currently executing Task)

If none of the sources for a variable's value is a child, then the value is currently 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 to potentially eliminate stalls. The key observation is that copy statements do not immediately need the value of a variable and therefore can be lazily evaluated. 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 currently available and it operates the same as the transfer function for $\mathrm{wr}(\texttt{x})$.

Enter Statement: When a source from a task flows across a back edge and reaches the enter statement for the same task, OoOJava must perform some bookkeeping. The problem is that variable sources name a dynamic instance of a task; when the enter statement to a task creates a new instance, previous instances of the same task age by one. The transfer function for the statement $\texttt{enter}(t_{\text{curr}})$ in Figure 2 increments the age of previous variable sources from t_{curr}. The analysis becomes finite by bounding the age by k; a variable source with the age 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. 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 `exit`(t_{curr}) statement finds variables with at least one child source and changes such variables to have exactly one source, the current task instance. The compiler generates code to copy the values from the child tasks 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. Recall 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 avoid the overhead of searching 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 `exit`(t_{curr}) transfer function: 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 its 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.

4.2 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_{curr} into three categories based on $M(\texttt{x})$ at the relevant program point:

Immediate Access: When all of the sources in $M(\texttt{x})$ are from the current task t_{curr}, its ancestors, or their siblings, the variable stores a currently available value. Therefore, the compiler simply generates normal code to access the variable `x` immediately.

Optimized Stall: When all of the sources in $M(\texttt{x})$ are from a single instance of a child t_{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_{child} retires and copy the value of `x` before generating a normal access to `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's value is currently available — in this case the code tracks (1) whether the variable has a value, (2) the actual value if it is available, and (3) the source of the value if the value is not currently available. Then code is generated just prior to the access of `x` to check whether the value is available or whether a task instance will provide the value and the task must stalled for it.

To complete code generation for variable accesses, the compiler generates bookkeeping code at the exit statement of each task. As mentioned in our discussion of 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.

At the entrance to each task, the compiler generates code to issue the task. Each task has a count of its unresolved dependences. This count is initialized to a bias value that is larger than the number of dependences and then updated using atomic operations. The issue code iterates over all of the sibling tasks that the issued task depends on. If the sibling task has not retired, the newly issued task is added to its forwarding list. When the sibling task retires, it decrements the dependence count of all tasks on its forwarding list. Finally, the code subtracts off the bias value minus the number of outstanding dependences. When the dependence count becomes zero, the task is dispatched for execution.

5. Heap Dependence Analysis

Heap dependences have long posed a challenge to automatically parallelizing code that manipulates data structures. OoOJava reasons about a heap access in terms of the *heap root* used to reach the accessed heap object; 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 task, or it is the first object along a heap path accessed by a parent task after the exit of a child task. In the latter case, we refer to the parent statement that accesses the variable as a potential *stall site* because the parent task may have to stall there for a child task to complete.

Two tasks can only have a heap dependence when both of the following two conditions are true. The first 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 the field f of an object allocated at site a. We call the pair of accesses potentially conflicting accesses and objects allocated at site a potentially conflicting objects. 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

OoOJava uses a static effects analysis extension to a standard pointer analysis to report 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 of the affected object, (3) the effect type (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 program reached the affected objects through the `AST` object referenced by the `ast` in-set variable of the 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 the same AST objects. We next describe a dynamic approach that can rule out such conflicts.

To issue an instance p of task `tpar`, the system would dynamically check for conflicts between p and all instances of tasks that have not retired: take, for example, an earlier instance p_0 of the task `tpar`. The runtime would traverse 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 that an effect could apply to. If this dynamic check shows that there are no conflicting accesses between the instance p and all previous non-retired task instances to the same object, then it is safe to immediately dispatch 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 `tpar` 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 summarizes the heap effects of a task. Section 5.3 presents an overview of 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 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

To be more precise, 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.

Recall that the analysis uses two different types of heap roots. The first type of heap root abstracts 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 stall site or in-set 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 define helper functions $E(\texttt{x}) = \{\langle \texttt{x}, n\rangle \in E\}$ and $E(\texttt{x},\texttt{f}) = \{\langle n, \texttt{f}, n'\rangle \in E \mid \langle \texttt{x}, n\rangle \in E\}$. We also assume that the pointer analysis provides a function $\mathcal{A}$ that maps a heap node to an allocation site. Though we assume a heap node only abstracts objects allocated at one site, modifications to support pointer analyses 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 an edge to the heap roots that were used to reach the edge's target.

The analysis also computes at each program point a set of variables $\mathcal{L}$ that the application may have to stall for before accessing the object they reference if the variables reference data structures for which there is a conflict.

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

Figure 3. Transfer Functions for Computing Heap Roots

The mapping R and set $\mathcal{L}$ both form lattices. The partial order ($\sqsubseteq$) is defined by the subset relation ($\subseteq$); join($\sqcup$) is set union ($\cup$); bottom($\bot$) is the empty set($\emptyset$); and top($\top$) is the maximally full set. The lattices have finite heights 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 the edges' target 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 roots for the corresponding variable's edges. Heap roots are then subsequently propagated to newly created references because we are interested in determining which heap root was used to access an affected object.

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 accesses to the objects referenced by 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 for child tasks. The transfer functions for $\mathcal{L}$ remove a variable at a statement that reads the

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

Figure 4. Transfer Functions for Effects

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 object fields in the heap and therefore are relevant for collecting effects. These transfer functions record for each field access: the heap root that was used to reach the object, the allocation site of the object, the operation, and the field that 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. Section 5.3.2 discusses the reasons for the strong update rule.

5.2.3 Interprocedural Extension

We next present the interprocedural extension to the effects analysis. The pointer analysis we used handles method calls by taking a snapshot of the portion of the caller's heap reachable by the callee just prior to the method invocation and labeling the heap elements with predicates—these predicates are initially tautologies. 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's 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 created in 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 Effect Conflict Elimination

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. Consider an task t_0 with the effect $\langle h_0, a_0^{\text{aff}}, o_0, f_0\rangle$ and a 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 $o_0 = o_1 = \texttt{read}$, 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 through heap references. 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 have the same heap node. The arity element of the pair is taken from the set $\{\texttt{0}, \texttt{1}, \texttt{MANY}\}$. A reachability state that abstracts the reachability of an object o that includes the pair $\langle n_1, \mu\rangle$ means that at most μ 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_n 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 $s \in S_n$ that conservatively abstracts the reachability of o.

The analysis also associates a set of reachability states S_e with each edge $e \in E$ in the points-to graph. Reachability states associated with an edge characterize the reachability of the objects reachable from that edge. Intuitively, these reachability states provide a precise way to propagate reachability changes through the heap abstraction. 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 $s \in S_e$ that conservatively abstracts the reachability of object o in the set of reachability states for each edge $e_1; e_2; ...; e_j$, where each edge e_i abstracts reference r_i. A complete presentation of disjoint reachability analysis is 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 $[\langle n_2, \texttt{1}\rangle]$ and $[\langle n_3, \texttt{1}\rangle]$. This set of reachability states 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 possible (by moving Line 7 to just after Line 5, for instance), then the reachability state $[\langle n_2, \texttt{1}\rangle, \langle n_3, \texttt{1}\rangle]$ 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_{AST}. Next assume that an `AST` object serves as a data structure root object for a collection of `TreeNode` objects

summarized by n_{TN}. Disjoint reachability information associated with n_{TN} is critical to OoOJava for deciding when to safely execute instances of task `tpar`. If the set of reachability states associated with n_{TN} includes a state with reachability tuple $\langle n_{\text{AST}}, \texttt{MANY} \rangle$ 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 $\langle n_{\text{AST}}, 1 \rangle$, meaning a given `TreeNode` object is reachable from at most one `AST` object. If 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 effects: $\langle \langle st_0, v_0, a_0^{\text{root}} \rangle, a^{\text{aff}}, o_0, f \rangle$ and $\langle \langle st_1, v_1, a_1^{\text{root}} \rangle, a^{\text{aff}}, o_1, f \rangle$. OoOJava inspects the set of reachability states S for each heap node associated with a^{aff} to decide whether the effects conflict:

1. If $a_0^{\text{root}} \neq a_1^{\text{root}} \wedge \forall s \in S.s \cap (\{\langle a_0^{\text{root}}, 1 \rangle, \langle a_0^{\text{root}}, \texttt{MANY} \rangle\} \times \{\langle a_1^{\text{root}}, 1 \rangle, \langle a_1^{\text{root}}, \texttt{MANY} \rangle\}) \neq \emptyset$, then there is no conflict. If there is no reachability state that contains both heap roots, then the two heap roots cannot access the same object allocated at allocation site a^{aff} and therefore the effects do not conflict.
2. If $a_0^{\text{root}} = a_1^{\text{root}} \wedge \forall s \in S.s \cap \{\langle a_0^{\text{root}}, \texttt{MANY} \rangle\} = \emptyset$, 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 basic nor reachability-based conflict detection eliminates a conflict, we say the effects have a coarse-grained conflict.

It is important to note that reachability states 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 states at different times. Extended effects analysis adds `strong` to the domain of possible effects and generates a strong update effect whenever a write effect decreases the reachability state 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 set of heap dependence queues implements the conflict graph efficiently at runtime. Section 5.4.2 discusses the queue properties and Section 5.4.3 gives the algorithm for finding a minimal set of heap dependence queues that covers the conflict graph. Section 5.4.4 presents the process for generating code to issue a task by inserting it into relevant heap dependence queues.

5.4.1 Conflict Graphs

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 task and 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 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.

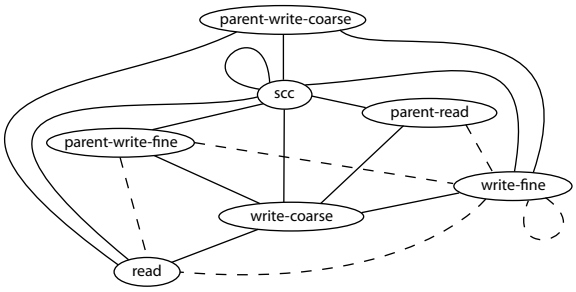

Figure 6. Heap Dependence Queue Ordering Constraints

5.4.2 Heap Dependence Queues

OoOJava compiles each conflict graph into a set of 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-conflicting-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 ordering constraint that accesses associated with the heap roots of the corresponding types can be only reordered if the heap roots refer to different objects. A solid edge indicates a coarse-grained ordering constraint that accesses associated with the heap roots of the corresponding types can never be reordered. Section 5.4.4 further discusses how we efficiently implement the heap dependence queues.

5.4.3 Compiling the Conflict Graph into Queues

We formulate compiling a conflict graph into a set of heap dependence queues as a graph covering problem. Each edge in the conflict graph must be enforced by an edge in a 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.

Figure 7 presents a greedy algorithm for heuristically solving the graph covering problem with a minimal set of heap dependence queues. The algorithm covers edges in the conflict graph until the given heap dependence queue cannot cover any more edges; the process repeats until every conflict edge in the conflict graph has been covered.

Line 2 creates a new heap dependence queue. Lines 3-13 cover the fine-grained edges in the conflict graph that the heap dependence queue will enforce. Line 3 checks for an uncovered fine grained edge. If an uncovered fine-grained edge is discovered, Line 4 adds one of its endpoints to the set U. The loop in Line 5 removes a node from the set U and the rest of the loop attempts to map the conflict graph node to a node in the heap dependence graph. Lines 6-9 try to map a node to a write node in the heap dependence queue, and Lines 10-13 try to map a node to a read node. A constraint on mapping a node in the conflict graph to a node in the heap dependence queue is that the mapping will not introduce a new ordering constraint that is not present in the conflict graph. Lines 6 and 10 verify that the mapping preserves this constraint.

Lines 14-16 perform the initial mapping of a node to a queue in the case that there was no fine-grained edge to cover. Line 17 computes the initial set of nodes that serve as end-points to coarse-grained edges that are incident to nodes mapped to the current

```
Cover(conflict graph C = (V, V_p, E, E_f, E_c))
1.  Set of covered edges E' ← ∅
2.  create new queue q ← (rd = ∅, wr = ∅, crs = ∅, par_rd = ∅,
                          par_wr = ∅, par_crs = ∅, scc = ∅)
3.  if ∃(u, v) ∈ E_f \ E' then
4.    U ← U ∪ {u}
5.    while U ≠ ∅ remove u from U
6.      if (u, u) ∈ E ∧ ∀x.QCovFine(q, x) ⟹ (u, x) ∈ E then
7.        E' ← E' ∪ {(u, v) | ∀v ∈ V, QCovFine(q, v)}
8.        U ← U ∪ {v | (u, v) ∈ E_f \ E'}
9.        if u ∈ V_p then par_wr ← par_wr ∪ {u} else wr ← wr ∪ {u}
10.     else if (u, u) ∉ E ∧ ∀x.QCovFnWr(q, x) ⟹ (u, x) ∈ E then
11.       E' ← E' ∪ {(u, v) | ∀v ∈ V, QCovFnWr(q, v)}
12.       U ← U ∪ {v | (u, v) ∈ E_f \ E'}
13.       if u ∈ V_p then par_rd ← par_rd ∪ {u} else rd ← rd ∪ {u}
14. else ∃(u, v) ∈ E_c \ E'
15.   if (u, u) ∈ E_c then scc ← {u}
16.   else if u ∈ V_p then par_rd ← {u} else rd ← {u}
17. U ← {u | ∀(u, v) ∈ E_c \ E', QCovFine(v)}
18. while U ≠ ∅ remove u from U
19.   if u ∈ V_p ∧ ∀x.QCovFnCh(q, x) ⟹ (u, x) ∈ E_c then
20.     E' ← E' ∪ {(u, v) | ∀v ∈ V, QCovFnCh(q, v)}
21.     U ← U ∪ {v | (u, v) ∈ E_f \ E'}
22.     par_crs ← par_crs ∪ {u}
23.   else if u ∉ V_p ∧ (u, u) ∉ E_c ∧ ∀x.QCovFine(q, x) ⟹ (u, x) ∈ E_c then
24.     E' ← E' ∪ {(u, v) | ∀v ∈ V, QCovFine(q, v)}
25.     U ← U ∪ {v | (u, v) ∈ E_c \ E'}
26.     crs ← crs ∪ {u}
27.   else if u ∉ V_p ∧ (u, u) ∈ E_c ∧ ∀x.QCov(q, x) ⟹ (u, x) ∈ E_c then
28.     E' ← E' ∪ {(u, v) | ∀v ∈ V, QCov(q, v)} ∪ {(u, u)}
29.     U ← U ∪ {v | (u, v) ∈ E_c \ E'}
30.     scc ← scc ∪ {u}
31. Q ← Q ∪ {q}
32. if ∃e ∈ E.e ∉ E' then goto 2 else return Q

QCov((rd, wr, crs, par_rd, par_wr, par_crs, scc), u) =
    (u ∈ rd ∨ u ∈ wr ∨ u ∈ crs ∨ u ∈ par_rd ∨ u ∈ par_wr ∨
     u ∈ par_crs ∨ u ∈ scc)

QCovFine((rd, wr, crs, par_rd, par_wr, par_crs, scc), u) =
    (u ∈ rd ∨ u ∈ wr ∨ u ∈ par_rd ∨ u ∈ par_wr)

QCovFnWr((rd, wr, crs, par_rd, par_wr, par_crs, scc), u) =
    (u ∈ wr ∨ u ∈ par_wr)

QCovFnCh((rd, wr, crs, par_rd, par_wr, par_crs, scc), u) =
    (u ∉ V_p ∧ (u ∈ rd ∨ u ∈ wr))
```

Figure 7. Conflict Graph Covering Algorithm

queue. Lines 18-30 then proceed to cover coarse-grained edges in the conflict graph by mapping conflict graph node to the current heap dependence queue.

5.4.4 Implementation

We next discuss the code OoOJava generates to issue a task. There is a record for each task that maintains a count of the task's unresolved dependences. When this count reaches zero, all of the task's dependences have been resolved and it can be safely dispatched. The count is updated using atomic operations — the thread that performs the atomic decrement that causes the count to reach zero is responsible for dispatching the task.

OoOJava first generates code to forward or copy the values of in-set variables to the new task as described in Section 4.2. OoOJava next generates code to preserve heap dependences. This code adds the newly issued task to each of the relevant heap dependence queues as determined by the conflict graph compilation algorithm.

OoOJava uses a similar code generation strategy for stall sites. At a stall site, a task adds itself to each heap dependence queue as determined by the conflict graph compilation algorithm. It passes a lock and a condition variable into each queue. It then waits to be notified by the heap dependence queue.

A potential issue arises with the read and write-fine states — it is possible that the value of the pointer in a variable will be forwarded by another task and is unknown when the task is issued. In this case, the heap dependence queue must maintain the total ordering of this task until it determines the actual value of the variable. It can then safely allow reorderings of the task.

The heap dependence queue manages heap dependences to determine which tasks may be safely executed out-of-order. Heap dependence queues are designed to provide constant-time operations. Heap dependence queues are implemented internally as a linked-list of hashtables for fine-grained entries (read, write-fine, parent-read, and parent-write-fine), vectors for coarse-grained entries (write-coarse and parent-write-coarse), and single entries for self-conflicting coarse-grained entries (self-conflicting-coarse).

A consecutive sequence of fine-grained entries in a heap dependence queues are grouped into a hashtable. The hash keys are unique object IDs that persist across garbage collections, and in the case of collisions, tasks are placed on a linked-list for that bin which preserves their execution order. Hash collisions can cause false dependences — we size the hashtable to be large enough that false collisions do not significantly limit parallelism relative to the number of processor cores that are available. This hash-based strategy enables OoOJava to respect fine-grained heap dependences with constant time hashtable operations.

Any consecutive sequence of coarse-grained entries 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 the coarse-grained elements.

All entries except parent entries remain in the queue after the corresponding task dispatches and are removed only when that task retires. A special property of parent entries is that they can be removed from the heap dependence queues as soon as their conflict is resolved. The observation is that a parent entry cannot conflict with later tasks as they have not been dispatched.

The self-conflicting-coarse entry may never be reordered with any other entries. Therefore, each task inserted with type self-conflicting-coarse is a single element in the heap dependence queue and nothing may be reordered across it.

6. Discussion

At this point, we have not discussed parallelizing programs that perform I/O. We present one strict and one relaxed strategy for handling I/O in OoOJava. The strict strategy is to serialize all I/O operations in the program. In this strategy, every I/O operation is serialized and the trace of I/O operations will match the serial elision. The strict strategy prohibits parallelism when unrelated tasks access disjoint sets of file descriptors.

We can relax our I/O strategy to expose additional parallelism if the application satisfies the following two conditions: (1) the developer annotates whether different file descriptors may access the same file and (2) external programs do not depend on the exact ordering of accesses across different files. This strategy is straightforward to implement in OoOJava — simply model native methods that implement file operations as writing to a special field in the file descriptor object. We expect that the relaxed strategy is sufficient for the correct behavior of many programs and has the potential to expose significantly more parallelism.

OoOJava can be extended to safely support applications that combine threads and tasks. The basic idea is that threaded programs could call into libraries that are implemented using tasks. In this model, the developer is responsible for using locks to ensure that the threaded parts of the program do not concurrently modify data structures accessed by tasks. OoOJava would then analyze the task part of the program in isolation assuming a maximally aliased heap at entry. Before exiting the task part of the program, the parent thread would stall until all tasks retire.

Benchmark	Measured Speedup	Lines
RayTracer	20.76×	2,832
Tracking	16.64×	4,747
Power	16.98×	1,846
KMeans	11.89×	3,220
Barnes-Hut	10.01×	3,162
Crypt	12.78×	2,035
MergeSort	12.34×	1,895
Labyrinth	10.31×	4,315
MonteCarlo	26.39×	5,669

Figure 8. Speedups on Benchmarks (Higher is Better)

Parallelization requires revisiting Java's exception model. It is helpful to divide exceptions into two categories: expected exceptions that developers write handlers for and unexpected exceptions that cause an application to simply exit. Expected exceptions that are caught inside the same task do not pose an issue for OoOJava. As a result, the common use of exceptions to recover from expected error conditions is largely unaffected by OoOJava.

Nearly all Java statements can potentially throw some type of unexpected exception (null pointer exception, array bound exceptions, division by zero). Developers typically do not write exception handlers for such exceptions and simply allow them to halt the program. Precisely handling such exceptions requires support for rollback because an application may execute past a task that later throws an uncaught exception. OoOJava gives such exceptions imprecise semantics to improve performance.

It is of course possible to efficiently support imprecise try-catch blocks that enclose a large number of dynamic tasks. Such blocks catch the exceptions of all enclosed tasks, but allow the enclosed tasks to be executed out-of-order. Such a block is implemented by simply waiting until all enclosed tasks have retired.

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 compiler generates C code which is then compiled by GCC. We enabled all optimization in GCC and classic compiler optimizations in our compiler. Our implementation and benchmarks are available on the web. We report times averaged over 10 runs.

7.1 Benchmarks

We selected a diverse set of benchmarks to provide an interesting cross-section of application behaviors and a variety of algorithmic structures and ported them to OoOJava. We took Crypt, RayTracer, and MonteCarlo from the Java Grande Benchmark suite [24]. We took Barnes-Hut from the Lonestar benchmark suite [16]. We took both KMeans and Labyrinth from the STAMP benchmark suite [9] to explore benchmarks with irregular parallelism. We took Power from the JOlden [8] benchmark suite and MergeSort from DPJ suite [6]. We took Tracking from SD-VBS [27].

7.2 Performance Discussion

Figure 8 summarizes the measured speedups and lines of code including libraries for each of our benchmarks. We report speedups relative to a sequential version compiled using the Java frontend of the same compiler.

RayTracer computes rows of a scene in parallel and executes a checksum operation that must be serialized; the speedup of 20.76 × indicates OoOJava is able to efficiently execute RayTracer across the available cores.

Tracking extracts motion information from a sequence of images. We targeted the most computationally expensive stage of the algorithm and obtained a 16.64× speedup.

KMeans, Power, and Barnes-Hut all execute many iterations of a parallel computation followed by a sequential computation. Though the repeated sequential portions of these benchmarks limit available parallelism, the speedups for these benchmarks are still significant. Our version of KMeans is 1.70× faster than the parallelized TL2 version included in the STAMP benchmark suite even though our version incurs significant additional overheads to implement array bounds checks. With array bounds checking disabled, our version is 2.62× faster than the parallelized TL2 version.

Crypt consists of an encryption phase and a decryption phase, both of which are followed by a sequential reconstruction of the full message. Although the sequential reconstruction limits available parallelism, the benchmark still achieves significant speedups.

The MergeSort taken from DPJ implements a sequential merge operation. This sequential code has a non-trivial overhead and limits parallelism at all levels of recursion.

Labyrinth is an interesting case of employing a speculative strategy in a deterministic environment. Each iteration launches an independent parallel routing computation, some of which may compute conflicting routes. In series, compatible routes are added to the master solution and conflicting routes are rescheduled for the next parallel iteration. Both the sequential computation and conflicts between routes limit available parallelism. OoOJava achieved a speedup of 10.31× for this speculative algorithm. Our version is 1.51× faster than the parallelized TL2 version included in the STAMP benchmark suite even though our version incurs significant additional overheads to implement array bounds checks. With array bounds checks disabled, OoOJava is 2.08× faster than the parallelized TL2 version.

The speedup for MonteCarlo reflects that the individual simulations are independent and that there is much parallelism.

To quantify the overhead of our research compiler, we compared the generated code against the OpenJDK JVM 14.0-b16 and GCC 4.1.2. The sequential version of Crypt compiled with our compiler ran 4.6% faster than on the JVM. We also developed a C++ version compiled with GCC and found our compiler's version ran 25% slower than the C++ version. Our compiler implements array bounds checking; with array bounds checking disabled, the binary from our compiler runs only 5.4% slower than the C++ binary. We used the optimization flag `-O3` for the C++ version as well as for the underlying C code generated by our compiler. This is close agreement with more extensive experiments on six benchmarks that we performed in earlier publications. Those experiments measured an average overhead for our compiler with array bounds checks disabled of 4.9% relative to GCC.

7.3 Parallelization Discussion

We typically began our parallelization efforts by using a profiler to identify computationally intensive sections of code. From there we manually inspected the hot spots and added task annotations to expose expected parallelism. The reported dependences informed us when the annotations did not result in the implementation we expected. In this way we incrementally parallelized a benchmark until the implementation matched our expectation of the available parallelism in the benchmark's core algorithms. Parallelization efforts typically required on the order of one to two dozen line changes.

Our experience porting RayTracer demonstrated the value of OoOJava's developer feedback. We enclosed the work for each row of pixels in a task and expected its instances to have no dependences; OoOJava reported the instances had conflicting accesses to a global scratch pad object. We simply moved the allocation of the scratch pad object into the task and obtained the implementation reported on above.

Labyrinth was more challenging. The original version of Labyrinth from STAMP intentionally used data races to read a shared map for route planning and then later used safe reads to verify that a planned route was valid. We modified the benchmark to maintain the same basic speculative algorithm, but in a safe, easy-

to-debug, deterministic form. Our modified version uses rounds of parallelized route planning followed by sequential code to apply the routes from the previous parallel round. Our version is both much simpler to debug and faster than the original STAMP version.

8. Related Work

Parallel functional languages [17] can offer strong correctness guarantees for parallel applications. However, these languages place onerous restrictions on the mutation of data structures, and therefore make it difficult to efficiently express many algorithms.

Kendo [18] addresses challenges of developing parallel software by enforcing deterministic interleavings for the widely used explicit thread and lock model. Models that generate parallel implementations from sequentially expressed code, like OoOJava, make reasoning about program behavior easier for the programmer.

Coarse-grained or macro-dataflow languages [11, 14] compose several sequential operations together to construct larger granularity code segments for dataflow execution. OoOJava can be viewed as a combined dynamic-static approach to translating imperative code into macro-dataflow code.

Deterministic, speculative models [29, 13, 28, 5] offer simple programming models, but incur potentially significant dynamic overheads to support recovery from mis-speculation and to dynamically check for potential conflicts. Static analysis enables OoOJava to parallelize applications without incurring these overheads.

Several non-speculative models including OpenMP [10], Cilk [20], or JCilk [12] rely upon correct developer annotations. Errors in these annotations can cause these models to silently produce incorrect results. Annotation errors in a OoOJava program never affect its correctness.

Other systems require extensive developer annotations to avoid unchecked access to data structures [22, 6] or additional code to create serialization sets [2]. OoOJava requires minimal annotations, which will likely improve developer productivity.

CellSs dynamically schedules function invocation when a function's operands are available [4]. CellSs does not perform heap dependence analysis and therefore it must restrict superscalar functions to pass-by-value and does not permit passing data structures that contain pointers. CellSs restricts superscalar execution to functions to avoid variable hazards; OoOJava's variable analysis and value forwarding allow tasks to be freely used wherever a developer finds it convenient.

OoOJava addresses parallelism opportunities that can be difficult for similar systems to take advantage of. For example, OpenMP, Cilk, and CellSs require explicit synchronization before accessing the results of parallel computations. Cilk with *inlets* and OpenMP with *reductions* do support limited aggregation of results in a parallel loop, however the aggregation operations in these systems do not necessarily execute in the same order as prescribed by the serial elision which would break non-commuting operations. Moreover, neither OpenMP nor Cilk can schedule an arbitrary chain of out-of-order computations, while OoOJava can.

OoOJava differs from inspector-executor approaches [21, 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.

Decoupled software pipelining (DSWP) [19] maps memory operations in a loop that may conflict to the same thread of a software pipeline. While this approach simplifies the necessary heap analysis, it limits parallelism — at most one thread can write to a statically identified heap region. In contrast, OoOJava can execute instances of write instructions across many cores. OoOJava also uses a sophisticated heap dependence analysis that can determine that some write statements of a loop are conflict-free where DSWP cannot. DSWP extracts very fine-grained parallelism compared to OoOJava; the techniques are likely synergistic.

In an earlier position paper [1] we explored an out-of-order software model without providing technical details. In this current work we describe in detail the necessary analysis and runtime support to implement OoOJava, and present a complete evaluation.

9. Conclusion

For parallel programming to become mainstream, parallel programming tools must become easy to use. We presented an approach to parallel programming that uses annotations to suggest parallelization of a sequential program. OoOJava automatically handles the details of implementing the parallelization and guarantees that the parallel version has the same behavior as the original sequential version. We have successfully parallelized nine applications and achieved significant speedups. Moreover, we found that parallelizing applications with OoOJava was straightforward and required only minor modifications to our benchmark applications.

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. A type and effect system for deterministic parallel Java. In *OOPSLA*, 2009.

[7] S. Borkar. Thousand core chips: A technology perspective. In *DAC*, 2007.

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

[11] K. Dai. Code parallelization for the LGDG large-grain dataflow computation. In *CONPAR 90 - VAPP IV*, 1990.

[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] C. Huang and L. V. Kale. Charisma: Orchestrating migratable parallel objects. In *HPDC*, 2007.

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

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

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

[18] M. Olszewski, J. Ansel et al. Kendo: Efficient deterministic multithreading in software. In *ASPLOS*, 2009.

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

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

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

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

[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.

[27] S. K. Venkata, I. Ahn et al. SD-VBS: The San Diego Vision Benchmark Suite. In *IISWC*, 2009.

[28] C. von Praun, L. Ceze et al. Implicit parallelism with ordered transactions. In *PPoPP*, 2007.

[29] A. Welc, S. Jagannathan et al. Safe futures for Java. In *OOPSLA*, 2005.