DOJ: Dynamically Parallelizing Object-Oriented Programs

Yong hun Eom Stephen Yang James C. Jenista Brian Demsky
University of California, Irvine
{yeom, stephey, jjenista, bdemsky}@uci.edu

Abstract

We present Dynamic Out-of-Order Java (DOJ), a dynamic parallelization approach. In DOJ, a developer annotates code blocks as tasks to decouple these blocks from the parent execution thread. The DOJ compiler then analyzes the code to generate heap examiners that ensure the parallel execution preserves the behavior of the original sequential program. Heap examiners dynamically extract heap dependences between code blocks and determine when it is safe to execute a code block.

Previous work on Out-of-Order Java used static analysis to allow code blocks to possibly execute out of order, similar to instructions in a super scalar processor. DOJ elides much of the complex static analysis in Out-of-Order Java.

We have implemented DOJ and evaluated it on twelve benchmarks. We achieved an average compilation speedup of 31.15× over OoOJava and an average execution speedup of 12.73× over sequential versions of the benchmarks.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis

General Terms Algorithms, Performance

Keywords Parallel Programming, Dynamic Analysis, Object-Oriented Analysis, Heap Analysis, Parallelization

1. Introduction

With the wide-scale deployment of multi-core processors and the impending arrival of many-core processors, software developers must write parallel software to realize the benefits of continued improvements in microprocessors. Developing parallel software using today's development tools can be challenging. Experience has shown that applications written in today's thread and lock-based model are prone to both data races and deadlocks.

Hardware has long benefited from extracting unstructured parallelism from sequential instruction streams through out-of-order execution [28]. Processors dynamically extract dependences between sequential instructions and then execute the instructions in parallel while preserving dependences. This paper leverages the same proven techniques at a coarser granularity to parallelize software.

Task-based dataflow programming models such as StarSs[11], OoOJava[17], and Sequoia[12] have recently emerged as a new parallel programming approach. These approaches implement the well-known out-of-order execution approach from hardware in software using tasks as the basic work unit. The approach can be viewed as a hybrid that executes a task-based von Neumann program using a dataflow execution model. In this model, tasks are *dispatched* by a sequential thread and begin executing when their dependences are resolved. When a task finishes execution, it *retires*. If a thread attempts to access data produced by one of its child tasks, the thread must stall until the child retires. Previous work on task-based dataflow programming models either required the developer to explicitly state dependences between tasks (StarSs) or used heavyweight static heap analysis (OoOJava).

The goal of this paper is to make the programming model from OoOJava practical by addressing the two primary technical challenges for deploying OoOJava in the real world. Scaling the reachability analysis used by OoOJava to large programs remains an open research problem. A major advantage of the DOJ approach is that it can achieve nearly the same runtime performance while relying only on standard pointer analysis. This enables DOJ to leverage recent work on pointer analysis (flow-sensitive pointer analysis has been scaled to over a million lines of code [13]). A second limitation of the static analysis approach taken in OoOJava is that it is hard for the developer to know when the analysis will extract sufficiently strong reachability properties to enable parallelization. The hybrid approach taken in this paper depends less on the precision of the static heap analysis; in DOJ, heap analysis imprecision typically results only in less optimized runtime checks.

In this paper we introduce a new approach to dynamically extract heap dependences between tasks. Our approach uses a static effects analysis to conservatively determine the possible heap effects of a task. While the results of the static effects analysis are not precise enough by themselves to parallelize many applications, they are sufficiently precise to build efficient, dynamic analyses that can precisely compute heap effects. A major advantage of this new approach is that the dynamic analyses can identify conflicts between tasks more precisely than a fully static analysis.

1.1 Basic Approach

DOJ respects all of the program dependences of the original sequential program. It does this by first categorizing dependences as either control dependences or data dependences. DOJ respects control dependences trivially by requiring all tasks to have exactly one exit. DOJ further divides data dependences into two categories: variable dependences and heap dependences. Variable dependences occur when one task writes to a variable and another task reads from that variable. DOJ uses the same value forwarding approach as OoOJava to eliminate write-after-write and write-after-read hazards on variables to enable parallelization. Heap dependences occur when one task writes to an object field and another task accesses that same field.

DOJ abstracts heap reads and writes using static *heap effects* expressed in terms of *heap roots*. We call heap effects that may introduce a heap dependence between two tasks *potentially conflicting effects*. A heap root is a variable that is live into a task and through which deeper heap references are obtained.

Heap roots occur in two contexts: a heap root is either a variable that is accessed by a task, or a variable that references the first object along a heap path accessed by a code following the exit of a task. In the latter case, we refer to the statement that accesses the variable as a potential *stall site* because the execution of that statement may have to stall until a previous task completes. DOJ uses static effects analysis to characterize the heap effects of a code block in order to produce a heap path and allocation site sensitive abstraction called an *effects finite state machine* (EFSM).

DOJ uses EFSMs to build *heap examiners* for each heap root. Heap examiners only traverse the fields that are necessary to compute the target objects for all potentially conflicting effects; this information is sufficient to dynamically detect the absence of heap dependences between tasks and enable parallelization. A heap root may potentially reference a very large data structure; traversing such data structures can incur large overheads. DOJ addresses this with lightweight but less precise dynamic checks based only on static heap effects to extract dependences between tasks. The less precise approach executes concurrently with heap examiners and limits the worst case overhead of traversing large data structures by ensuring that tasks never run significantly slower than the sequential code. Once a task is dispatched, the task is guaranteed to have no conflicts and runs to completion without any extra overhead.

DOJ has four primary advantages over the static approach taken by OoOJava: (1) its dynamic analysis is, in general, more precise than the previous static analysis, (2) its dynamic analysis is sensitive to the actual fields traversed by tasks and not just reachability through any path, (3) the dynamic analysis can more easily scale to large code bases, and (4) DOJ can determine that non-conflicting updates to the same data structure can run in parallel.

1.2 Contributions

This paper makes the following contributions:

- **Heap Examiners:** It presents heap examiners, a statically-directed dynamic analysis for predicting the heap effects of a code block before it runs. Heap examiners improve the precision of the static effects analysis to enable DOJ to effectively parallelize our benchmarks.
- **Optimizations:** It presents a set of pruning optimizations that reduce the overhead of heap examiners.
- **Hierarchical Approach to Heap Dependences:** It presents a hierarchy of approaches for detecting heap dependences. The hierarchy includes both (1) a precise approach to determine heap dependences to extract parallelism and (2) a fast and imprecise approach to limit the worst case dynamic overheads.
- **An Implementation and Evaluation:** We have implemented DOJ and evaluated its performance on twelve benchmarks.

2. Example

Figure 1 presents an example DOJ program. The loop in Line 9 of the example creates one hundred `Foo` data structures and inserts them into a set. The loop in Line 16 iterates over `Foo` data structures in the set, calls the `compute` method on each `Foo` data structure, and then sums the return values from the calls to `compute`.

DOJ extends the sequential Java programming model in the same way as OoOJava — it adds the *task* annotation to the sequential Java programming model, which tells the compiler that the code block enclosed in the task should be decoupled from the parent thread's execution and be executed when its dependences are resolved. Tasks may be nested and may contain arbitrary code, with the exception that tasks have a single exit. We call a task nested within another task—possibly in another context—a child/parent task relation. If the parent thread performs an operation that may conflict with or use data from one of its child tasks, it must stall until that task retires. It is important to note that task annotations never affect the semantics of the program — DOJ guarantees that the execution always preserves the sequential semantics of the unannotated program.

Calls to the `compute` method in Line 20 from the same iteration of the outer loop in Line 15 operate on disjoint data and therefore can execute in parallel. However, calls from different iterations of the outer loop may have data conflicts. We added the task declaration in Line 19 to allow calls to the `compute` method to execute (possibly out-of-order) when their data dependences are resolved. Tasks have names for pedagogical convenience; task names have no semantic meaning. The summation in Line 23 has both a dependence on the `compute` method from the same loop iteration as well as the `sum` variable's value from the previous loop iteration. We added the task declaration in Line 22 to allow the loop to execute past the summation to dispatch additional `par` task instances.

We note that programming models like Cilk [22] have an explicit reduction construct for code like Line 23. Reductions should be commutative operations and may commit out of order, which breaks sequential semantics but may expose additional parallelism. DOJ trades this parallelism opportunity to make a strong guarantee: task annotations never change the sequential semantics, which lets developers reason in a single-threaded model.

We also note that the iteration over a set in Line 16 defeats most static approaches to automatic parallelization. Automatic static parallelization of this loop is difficult because it requires extracting

```
1  public class Foo {
2    Bar cntr;
3    Bar inc;
4    public Foo(Bar cntr, Bar inc) {
5      this.cntr=cntr; this.inc=inc;
6    }
7    public static void main(String x[]) {
8      HashSet set=new HashSet();
9      for(int i=0; i<100; i++) {
10       Bar cntr=new Bar(i);          //Allocation site 1
11       Bar inc=new Bar(1);           //Allocation site 2
12       set.add(new Foo(cntr, inc)); //Allocation site 3
13     }
14     int sum=0;
15     for(int j=0; j<10; j++) {
16       for(Iterator it=set.iterator(); it.hasNext(); ) {
17         Foo f=(Foo) it.next();
18         int val;
19         task par {                  //Parallelizable task
20           val=f.compute();
21         }
22         task seq {                  //Sequential task
23           sum+=val;
24         }
25       }
26     }
27     System.out.println("Total:"+sum);
28   }
29   public int compute() {
30     return cntr.value+=inc.value;
31   }
32 }
33
34 public class Bar {
35   public Bar(int value) {
36     this.value=value;
37   }
38   public int value;
39 }
```

Figure 1. DOJ Example

complex properties about the behavior of the set implementation. The approach for identifying task dependences in DOJ generalizes to any control structure: counted loops, data-dependent loops, recursive tasks, etc.

2.1 Data Dependences

Recall that DOJ splits data dependences into two categories: variable dependences and heap dependences. A variable dependence occurs when one task writes to a variable and another task reads from that variable. The tasks in the example contain several variable dependences. The `par` task in Line 19 has a read dependence on the parent thread for the reference stored in the variable `f` and writes a value to the variable `val`. A `seq` task instance from Line 22 has a read dependence on the value written to the variable `val` by the `par` task instance from the same loop iteration. A `seq` task instance has a read dependence on the value written to the variable `sum` by the previous instance of the `seq` task. The parent thread has a dependence in Line 27 on the value of the `sum` variable written by the last instance of the `seq` task. DOJ's variable dependence analysis automatically discovers these variable dependences.

Heap dependences occur when one task instance writes to the field of an object and another task instance accesses the same object field. There are heap dependences between instances of the `par` task in the example. Each instance of the `par` task reads and updates the `value` field of a `Bar` object and then obtains a reference to this `Bar` object by following the `cntr` field of the `Foo` object referenced by the variable `f`. Most static heap analyses would determine that instances of the `par` task update the `value` field of objects allocated at allocation site 1, and therefore may conflict with each other and cannot be safely parallelized.

2.2 Abstracting Heap Effects

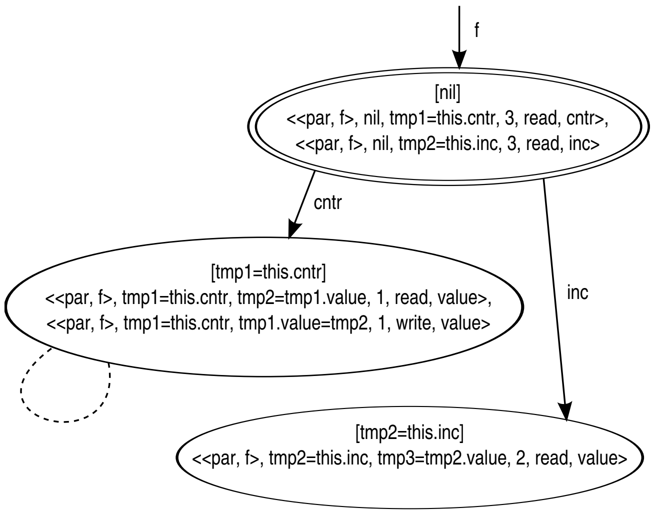
f
[nil]
<<par, f>, nil, tmp1=this.cntr, 3, read, cntr>,
<<par, f>, nil, tmp2=this.inc, 3, read, inc>
cntr
inc
[tmp1=this.cntr]
<<par, f>, tmp1=this.cntr, tmp2=tmp1.value, 1, read, value>,
<<par, f>, tmp1=this.cntr, tmp1.value=tmp2, 1, write, value>
[tmp2=this.inc]
<<par, f>, tmp2=this.inc, tmp3=tmp2.value, 2, read, value>

Figure 2. Initial EFSM for Task `par`

DOJ uses effects finite state machines (EFSMs) to abstract the heap effects of tasks and code blocks between tasks. EFSMs are both heap path and allocation site sensitive — for each heap effect, the EFSM captures the heap path that the application code uses to reach the affected object and the allocation site of the affected object. Heap path sensitivity implies EFSMs are execution path sensitive as well; intuitively an EFSM abstracts the code of task with respect to heap accesses.

EFSMs are compiled into heap examiners — heap examiners perform the heap traversal abstracted by the EFSM to compute the set of affected objects for each static heap effect.

The initial state in an EFSM corresponds to the task entrance or stall site, and the other states correspond to dereference statements in the code an EFSM abstracts. States are annotated with *effects tuples* that abstract the heap effects the task performs on objects abstracted by the state. An edge abstracts the action of dereferencing an object to obtain references to further objects — the edge begins at the statement that obtained a reference to the object to be dereferenced and ends at the dereference statement that reads the reference from that object's fields. The edge is labeled with the field or variable that was read to obtain the new object reference. Edges therefore capture the heap paths that tasks can potentially traverse through the heap.

Figure 2 presents the initial EFSM for the `par` task. The edge labeled `f` directed into the initial state indicates that the initial object reference is obtained by reading the variable `f`. The annotation `[nil]` indicates that this state abstracts object references at the task entrance. The effect annotation on the initial state, $\langle\langle \texttt{par}, \texttt{f}\rangle, \texttt{nil}, \texttt{tmp1} = \texttt{this.cntr}, 3, \texttt{read}, \texttt{cntr}\rangle$, indicates that object references abstracted by this state that were allocated at allocation site 3 may flow to the statement `tmp1=this.cntr`, and that statement may then read the `cntr` field of those objects. The outgoing edge labeled `cntr` shows that this effect takes an object reference obtained at the task entrance, reads its `cntr` field, and makes the object referenced by the `cntr` field available at the exit of the dereference statement `tmp1=this.cntr`. This edge abstracts the effect of the `cntr` dereference that appears in the `compute` method that is called by the `par` task. The two effects on the edge's destination state abstract the read and write of the `value` field in Line 30.

The presence of a dashed line between two states indicates a potential conflict between those states. For example, the dashed line on the state `[tmp1=this.cntr]` indicates that effects represented by that state can conflict with themselves.

2.3 Optimizing EFSMs

DOJ compiles EFSMs into heap examiners that dynamically compute the heap effects of a task by performing the traversal abstracted by the EFSM. Therefore, edges in an EFSM have a runtime cost — the heap examiner must traverse these edges to compute the set of objects that a task may affect. In many cases, the compiler can statically determine that some of the heap effects in an EFSM can never introduce runtime heap dependences between tasks. Our compiler uses a set of rules to prune irrelevant heap effects from EFSMs. The compiler uses the type of effect (read or write), the allocation site of the affected object, and the affected field to compute whether a given heap effect can conflict (i.e., introduce heap dependences between tasks) with any other heap effect. If a heap effect can never conflict, we can safely prune that heap effect from the EFSM. If an edge in an EFSM does not lead to any conflicting heap effects, there is no reason to traverse that edge and it can be safely pruned from the EFSM, as described in Section 4.4.3. We describe in Section 4.4.4 how our compiler can merge redundant EFSM states to further optimize the heap examiners.

Figure 3 presents the result of pruning the EFSM in Figure 2. The examiner corresponding to this effects graph no longer inspects the `Bar` object referenced by the `inc` field because the compiler has determined that the `Bar` object is only read, so the `inc` reference does not lead to a potentially conflicting heap effect.

2.4 Runtime Checks

DOJ compiles the optimized EFSMs into C code that computes the set of affected objects for each potentially conflicting heap

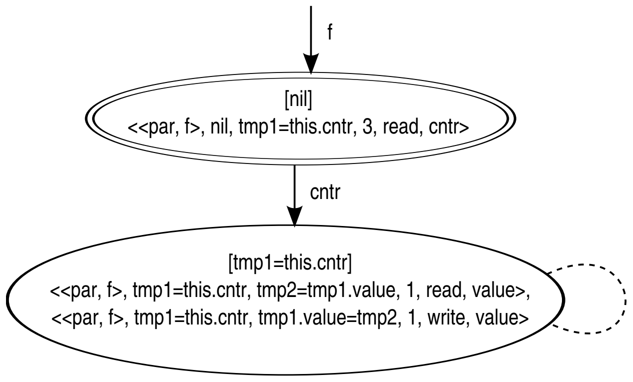
f
[nil]
<<par, f>, nil, tmp1=this.cntr, 3, read, cntr>
cntr
[tmp1=this.cntr]
<<par, f>, tmp1=this.cntr, tmp2=tmp1.value, 1, read, value>,
<<par, f>, tmp1=this.cntr, tmp1.value=tmp2, 1, write, value>

Figure 3. Pruned EFSM

effect. For the example, the heap examiner for the current `par` task instance would first find the `Foo` object referenced by variable `f`, then follow that object's `cntr` field to a `Bar` object. This `Bar` object is compared to the `Bar` objects of all previous instances of the `par` task that have not retired. If the `Bar` object for the current `par` task is unique in this collection, the current task can be executed immediately in parallel with all other outstanding `par` task instances.

DOJ combines the precise dynamic check that traverses the heap with a lightweight, coarse-grained conflict detection approach that detects whether two task instances can conflict based only the static task effects. Both approaches execute concurrently and either approach can detect the absence of heap conflicts.

3. Variable Dependence Analysis

DOJ handles variable dependences using the same approach as OoOJava. In this section, we briefly outline this approach for completeness. A detailed explanation can be found in the OoOJava paper [17]. The basic approach forwards variable values to eliminate variable anti-dependences to enable multiple instances of the same task to execute in parallel.

Recall that a task has a variable dependence on an earlier task if it reads a variable that was last written to by the earlier task. The variable dependence analysis extracts the variable dependences between tasks. 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) the variable to which the task wrote the value. Variable source tuples statically characterize how the program's execution propagates values in variables between tasks. An intra-procedural, data-flow analysis computes the variable source tuples for every live variable at every program point. Variable source tuples provide the necessary information to route the values of variables between tasks.

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 the analysis results at the relevant program point:

Immediate Access: When all of the source tuples are from the current task instance t_{curr}, its ancestors, or their siblings, the variable currently stores the actual value. Therefore, the compiler simply generates normal code to access the variable `x` immediately.

Optimized Stall: When all of the source tuples are from a single instance of a child t_{child}, then the compiler knows which dynamic task instance will provide the value for variable `x`. In this case the generated code will stall the current task until t_{child} retires and copy the value of `x`. If the same task can be determined to be the source of other variables, the compiler generates code to also read those values. This optimization avoids extra dynamic checks for future accesses to those variables.

Dynamic Tracking: Otherwise, the variable dependence analysis cannot track the source statically. In this case, the compiler identifies the program points at which the statically known variable sources became unknown. The compiler inserts code at these points to dynamically track which task generates the variable's value.

4. Heap Examiners

Heap dependences pose a challenge for automatically parallelizing code that manipulates data structures. DOJ reasons about a heap access in terms of (1) the heap root used to reach the accessed heap object and (2) the path taken through the heap from the heap root to the affected object. Two tasks can have a heap dependence only when both of the following two conditions are true: (1) 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 and (2) there must exist a *potentially conflicting object*. A potentially conflicting object is reachable from both tasks by starting from a heap root and following only heap references allowed by statements in the task. We call such a pair of accesses *potentially conflicting accesses*.

4.1 Overview of Approach

Heap examiners inspect the heap prior to executing a task to compute the task's heap dependences. A heap examiner walks the same heap that the task would walk to identify potentially conflicting objects. One challenge is that a task may mutate the heap and then walk across newly created references. The heap examiner cannot mutate the heap, therefore the compiler analysis must convert the task's traversal of the heap into a traversal of the heap as it existed at the beginning of the task.

DOJ calculates heap dependences of a task by checking the set of affected objects against objects that may be accessed by any previously dispatched tasks that have not yet retired. A task may be dispatched as soon as the affected objects identified by its heap examiner will no longer be accessed by any previous task.

To construct a heap examiner, it is possible to simply use a points-to graph to compute all paths to the affected objects, but this approach would result in an overly conservative set of affected objects and unnecessarily incur extra runtime overheads for traversing irrelevant paths. Moreover, consider a relatively common pattern in which multiple tasks update disjoint, localized sections of a data structure (e.g., BarnesHut). Computing affected objects without considering access paths would result in heap examiners that cannot determine that the tasks' updates are disjoint.

DOJ uses static effects to build EFSMs that in turn are compiled into heap examiners. Section 4.2 presents the static heap effects analysis, Section 4.3 presents a method for checking conflicts statically, and Section 4.4 describes how our implementation translates static heap effects into EFSMs that conservatively approximate a task's heap effects dynamically.

4.2 Heap Effects Analysis

While the effects analysis is similar in some aspects to the effects analysis used in OoOJava, the effects analysis for DOJ must extract much more fine-grained information that captures the heap path used to obtain an object reference in order to build an EFSM from a program's heap accesses.

4.2.1 Analysis Domains

DOJ represents an effect as a 6-tuple $\langle h, \mathtt{st}^{\text{from}}, \mathtt{st}^{\text{curr}}, a^{\text{aff}}, o, f \rangle \in U \subseteq H \times ST \times ST \times A \times O \times F$, where h is the heap root used to access the affected object, $\mathtt{st}^{\text{from}}$ is the field or array dereference statement that provided a reference to the affected object, $\mathtt{st}^{\text{curr}}$ is the program statement that produced the effect, $a^{\text{aff}} \in A$ is the allocation site of the affected object, $o \in O = \{\mathtt{read}, \mathtt{write}\}$ is the operation, and $f \in F$ is the affected field.

Recall that the analysis uses two types of heap roots. The first type of heap root abstracts the objects referenced by a task's input variables (variables live into the task that are accessed by the task). The second type abstracts objects referenced by live variables in sections of a task following the exit of a child task. Formally, a heap root h is given by the tuple $\langle \mathtt{st}, v \rangle \in H \subseteq ST \times V$, where $\mathtt{st}$ is either a stall site or a task entrance, V is the set of variables in the program, and v is the stall site variable or input variable.

The analysis assumes the presence of a pointer analysis. Our implementation uses a flow-sensitive, field-sensitive pointer analysis. We believe that field-sensitivity is important for most Java applications. However, flow-insensitive pointer analysis should be sufficient for the general techniques from DOJ. We assume the pointer analysis abstracts objects with a set of heap nodes $n \in N$ and heap references with a set of edges (points-to set) $e \in E \subseteq V \times N \cup N \times F \times N$. We define helper functions $E(\mathtt{x}) = \{\langle \mathtt{x}, n \rangle \in E\}$ and $E(\mathtt{x}, \mathtt{f}) = \{\langle n, \mathtt{f}, n' \rangle \in E \mid \langle \mathtt{x}, n \rangle \in E\}$. We assume that the pointer analysis provides a function $\mathcal{A}$ that maps heap nodes to allocation sites. Though we assume a heap node abstracts objects allocated at only 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 $R \subseteq E \times H \times ST$ from an edge to both (1) the heap root and (2) the last dereference statement in the sequence of dereferences that were used to reach the edge's target. At each program point, the analysis also computes a set of variables $\mathcal{L}$ for which the application may have to stall before accessing the object they reference. The set $\mathcal{L}$ only includes variables if they reference data structures with conflicts.

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.

4.2.2 Transfer Functions

For pedagogical purposes, we decompose the effects analysis into two passes. The first pass computes the mapping R from edges in the points-to graph 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. Our actual implementation integrates both analyses into the pointer analysis implementation. Figure 4 presents the transfer functions for computing the mapping R from edges 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 to pre-compute which variable accesses may require stalls. This supporting analysis is relevant for sections of code following the exit of a 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 a stall to wait for heap dependences on child tasks to resolve. Figure 4 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 variable and therefore serves as a potential stall site for the data structure referenced by the variable.

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

Figure 4. Transfer Functions for Computing Heap Roots

`st`	$U' = U \cup GEN$
`x=y.f`	$\text{GEN} = \{\langle h, \mathtt{st}^{\text{from}}, \mathtt{st}^{\mathtt{x=y.f}}, \mathcal{A}(n), \mathtt{read}, \mathtt{f} \rangle \mid \forall \langle \mathtt{y}, n \rangle \in E,$ $\langle \langle \mathtt{y}, n \rangle, h, \mathtt{st}^{\text{from}} \rangle \in R_a\}$
`x.f=y`	$\text{GEN} = \{\langle h, \mathtt{st}^{\text{from}}, \mathtt{st}^{\mathtt{x.f=y}}, \mathcal{A}(n), \mathtt{write}, \mathtt{f} \rangle \mid \forall \langle \mathtt{x}, n \rangle \in E,$ $\langle \langle \mathtt{x}, n \rangle, h, \mathtt{st}^{\text{from}} \rangle \in R_a\}$

Figure 5. Transfer Functions for Generating Effects

Figure 5 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. Note that the effects analysis treats array operations as normal field accesses on a special array field.

4.2.3 Interprocedural Extension

Our implementation contains an interprocedural extension to the heap roots and effects analyses. The analysis propagates the heap root annotations from the caller to the callee. The analysis merges heap root annotations from all callers to a method and analyzes a single context. When mapping analysis results from a callee back to the caller context, the analysis uses a call graph to identify and remove heap root annotations that are impossible in the given caller context. This process preserves analysis precision by preventing the erroneous propagation of heap roots from one caller to another, while avoiding the cost of analyzing multiple caller contexts.

4.3 Static Conflict Detection

DOJ considers a task to have a heap dependence on an earlier task when both of the following two conditions are true: (1) the two tasks are siblings or a parent stall site/child task pair and (2) the effects of the two tasks conflict. Note that we attribute all of a child task's effects to its parent, which confines the possible dependence relations to those identified in (1). This simplifies static conflict detection and dynamic dependence tracking.

Consider a task t_0 with the effect $\langle h_0, \mathtt{st}_0^{\text{from}}, \mathtt{st}_0^{\text{curr}}, a_0^{\text{aff}}, o_0, f_0\rangle$ and a task t_1 with the effect $\langle h_1, \mathtt{st}_1^{\text{from}}, \mathtt{st}_1^{\text{curr}}, a_1^{\text{aff}}, o_1, f_1\rangle$. DOJ conservatively assumes all such effects conflict with the following exceptions:

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.

4.4 Dynamic Conflict Detection

Imprecision in static effects analyses often makes them insufficient to parallelize programs. This imprecision arises because their abstractions collapse many objects into a single static heap node. Consider two instances of the same task that update disjoint objects. The updated objects are likely to be represented by the same static heap node and therefore the static analysis results do not contain enough information to determine that the two task instances do not conflict. In addition to directly using the static analysis results to detect possible conflicts, DOJ uses the results of the effects analysis to generate heap examiners. Heap examiners traverse the heap reachable from a task's heap roots to compute at runtime the set of concrete objects that are the target of statically identified effects. The runtime is able to resolve conflicts with better precision than static analysis by computing the concrete targets of the effects.

4.4.1 Generating the Effects Finite State Machine

DOJ generates a heap examiner for each heap root in the program. This process begins by generating an EFSM. The EFSM captures all of the effects of the task on the part of the heap reachable from the given heap root. The compiler begins to generate an EFSM for a heap root by first collecting all of the effects for the given heap root. We formalize the EFSM as a collection of states $\phi \in \Phi$, and a collection of transitions between the states $\langle \phi^{\text{from}}, f, \phi^{\text{to}}\rangle \in T \subseteq \Phi \times F \times \Phi$. The effects map $\mathcal{M} \subseteq \Phi \times U$ maps states in the EFSM to the set of effects associated with that state.

The compiler initially generates a state in the EFSM for each program statement that appears in an effect in the heap root's effect set and a special initial state, *nil*. We formalize the initial mapping of program statements to states with the function $\mathcal{S} : ST \to \Phi$.

For each effect $\langle h, \mathtt{x = y.f}, \mathtt{st}^{\text{curr}}, a^{\text{aff}}, o, f\rangle$, if the field f references an object, the compiler adds the transition $\langle \mathcal{S}(\mathtt{x = y.f}), f, \mathcal{S}(\mathtt{st}^{\text{curr}})\rangle$ to the EFSM. Note that if the field f stores a primitive, the compiler does not add a transition to the EFSM. In either case, the compiler then adds the effects mapping pair $\langle \mathcal{S}(\mathtt{x = y.f}), \langle h, \mathtt{x = y.f}, \mathtt{st}^{\text{curr}}, a^{\text{aff}}, o, f\rangle\rangle$ to the map $\mathcal{M}$.

4.4.2 Computing Static Conflicts

The compiler next computes the set of potentially conflicting effects C for each EFSM. The computation begins by identifying all potentially concurrently executing EFSMs (including other instances of itself). Potentially concurrently executing EFSMs are those that share a parent task; the parent task of a stall site EFSM is the task in which the stall site appears. An exception is that two stall site EFSMs with the same parent can never execute simultaneously and therefore can never conflict.

For each effect e in the EFSM, the analysis looks for a potentially conflicting effect in a potentially concurrently executing EFSM. If such an effect is found, the effect e is added the EFSM's set of conflicting effects C.

4.4.3 Pruning the EFSM

The EFSM will be used to generate a heap examiner that computes at runtime the exact concrete objects that a given effect could apply to. An EFSM may contain effects that the analysis determines can never cause a conflict. In this case, the heap examiner does not need to compute which objects such an effect could apply to. Pruning such effects from the EFSM is beneficial as it optimizes the corresponding examiner by removing extraneous work.

DOJ applies the following rules to prune EFSMs until no rule applies:

1. Irrelevant Transitions: If there is no path from a given transition to some state with a potentially conflicting effect, then the code block will never conflict on an object that is reached through this transition. The compiler prunes such transitions.

2. Irrelevant Effects: If an effect has no conflicts and no corresponding transition, the effect is irrelevant and therefore the compiler prunes it.

4.4.4 Merging States in the EFSM

The algorithm as described can generate EFSMs with multiple transitions that each read the same field of an object in the same state. While it is possible that such transitions can improve the precision of heap examiners, we expect that they typically serve only to add additional runtime overhead. The compiler therefore includes a merging phase that merges extraneous states with the goal of reducing runtime overhead.

When the analysis identifies pairs of transitions that originate from the same state and read the same field of objects from the same allocation state, it merges the destination states of those transitions.

4.4.5 Compiling EFSMs

DOJ compiles the pruned EFSMs into heap examiners. We begin our presentation with the basic compilation approach and will later discuss optimizations. Examiners are structured as a graph traversal with a `tovisit` queue that stores objects to be visited and a `discovered` set that stores which objects have been discovered. The body of the loop is a `switch` statement on the state of the object, with a case for each state in the pruned EFSM. Each case contains a nested `switch` statement on the allocation site of the object, with a case for each allocation site in the given state of the EFSM. The case for an allocation site examines each field for which the state has an outgoing edge for the given allocation site.

```
 1 void parExaminer(Object f) {
 2   Pair pinit=new Pair(f, INITSTATE);
 3   tovisit.push(pinit);
 4   discovered.add(pinit);
 5   while(!tovisit.isEmpty()) {
 6     Pair p=tovisit.pop();
 7     switch(p->state) {
 8       case INITSTATE:
 9         switch(p->obj->allocSite) {
10           case 3:
11             Object cntr=p->obj->cntr;
12             Pair pcntr=new Pair(cntr, CNTRSTATE);
13             if (cntr!=NULL &&
14                 !discovered.contains(pcntr)) {
15               discovered.add(pcntr);
16               tovisit.push(pcntr);
17             }
18             break;
19         }
20         break;
21       case CNTRSTATE:
22         switch(p->obj->allocSite) {
23           case 1:
24             addWriteEffect(p->obj);
25             break;
26         }
27         break;
28     }
29   }
30 }
```

Figure 6. Examiner for the Heap Root `f`

Figure 6 presents a heap examiner for the pruned effects graph from Figure 3. The case statement in Line 10 examines the `cntr` field of objects in the initial state from allocation site 3. The case statement in Line 23 processes the write effect for objects from allocation site 1.

Tree Structure Optimization While the heap examiner compilation strategy must support data structures with arbitrary possibly-cyclic structures, many data structures contain components with tree-like structures. DOJ optimizes traversals of these tree-like components. We note that all cycles in an EFSM must contain at least one state with two or more incoming edges. The traversal loop only contains case statements for heap nodes that are either (1) referenced directly by the heap root variable or (2) have two or more incoming edges. From each such heap node, the compiler inlines the traversal code for the part of the EFSM that is transitively reachable from that node through nodes with at most one incoming reference. One side effect of this optimization is that an object can potentially be traversed more than once, for example when every element of an array references a single object, however such duplication is safe and the traversal is guaranteed to terminate.

Figure 7 presents the tree-optimized EFSM for the example. The traversal of objects allocated at site 1 has been inlined into the case statement for objects allocated at site 3.

Heap Examiner Conflicts Heap conflicts can cause naïve heap examining strategies to traverse the wrong set of objects. Figure 8 illustrates the potential issue. Consider three tasks given in execution order: the first with the heap root `f`, the second with the heap root `g`, and the third with the heap root `h`. The heap examiner for the second task could potentially read the wrong reference from the `x` field of the `Baz` object if it started before the first task finished. Reading the wrong reference could cause the heap examiner to wrongly conclude that the second and third task do not have a conflict even if they access the same instance of class `Foz`.

If a heap examiner traverses a reference for which it has a potential read conflict on the specific object, the examiner must

```
void parExaminer(Object f) {
  Pair pinit=new Pair(f, INITSTATE);
  tovisit.push(pinit);
  discovered.add(pinit);
  while(!tovisit.isEmpty()) {
    Pair p=tovisit.pop();
    switch(p->state) {
      case INITSTATE:
        switch(p->obj->allocSite) {
          case 3:
            Object cntr=o->cntr;
            if (cntr!=NULL && cntr->allocSite==1)
              addWriteEffect(o);
            break;
        }
        break;
    }
  }
}
```

Figure 7. Tree Optimization of the Examiner for the Heap Root `f`

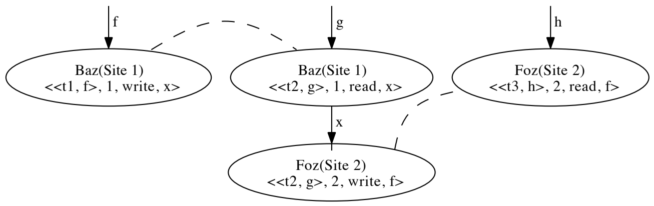

Figure 8. Heap Examiner Conflicts

stall until the conflicting task instance retires. When necessary, the compiler generates code to implement a heap examiner stall.

A heap examiner can also conflict with its own task instance if that instance begins executing before its heap examiner terminates (this can occur if the static check clears the task instance to execute). Specifically, if a task instance (1) reads from a reference field and uses the reference to perform potentially conflicting accesses (so the heap examiner will traverse the field) and (2) later overwrites the reference field, then the task instance can conflict with its own heap examiner. Our compiler statically identifies such dereferences in the EFSM and then generates a check after the dereference to verify that the task instance has not started executing. If the task has already started, the heap examiner stops traversing the heap and stalls to wait for the task to retire.

Weakly-Connected Components Extension The traversal of heap roots can be parallelized by separating the pruned EFSMs into weakly-connected components. This enables two optimizations: turning off individual heap examiners in favor of static resolution and parallelizing the heap examiners.

5. Runtime System

DOJ is architected as a set of *worker threads* that each contain a local work queue and implement a work stealing scheduling strategy. The DOJ compiler converts the entire program into tasks, with a wrapper task for the program's `main` method. DOJ supports hierarchical composition of tasks — tasks can dispatch other tasks. We refer to a worker thread that is currently executing a task that dispatches child tasks as a *parent thread*.

DOJ uses two approaches in parallel to determine whether a task may perform heap accesses that conflict with previously dispatched but not retired tasks. The first approach performs a dynamic object traversal to precisely determine conflicts, but can potentially take a long time. There is a companion *heap examiner thread* that implements this approach for each parent thread. The heap examiner thread examines the actual heap to determine whether the current task conflicts with previous dispatched, but not retired tasks. The heap examiner thread uses a *heap scoreboard* to detect potential conflicts between tasks. The second approach is based on static analysis results and is imprecise, but extremely fast. Each parent thread implements this approach with a *static effects queue* that uses static heap effects as determined by the compiler to detect conflicts between tasks. A task no longer has a conflict when its static effects queue reports all previously dispatched tasks with potential conflicts have retired. If either approach shows the absence of a potential conflict, DOJ dispatches the task. Intuitively, the combination of these two approaches works well because when the heap examiner resolves conflicts quickly the system can expose the parallelism, while static effects queues limit the worst case overheads for dynamically checking very large data structures.

5.1 Task Records

Each task instance has a *task record*. Each task record has a `NumUnresolvedDependences` count of the task's unresolved dependences — when this count reaches zero all of the task's dependences have resolved and the task can be safely executed. This count is updated by an atomic subtract instruction — updates to the count never need to obtain a lock. There is a structure in the task record for each heap root. This structure contains a `QueueDependences` count that tracks how many static queues the task record must clear to show the absence of a conflict for the given heap root. The structure also contains the `ObjDependences` count that tracks how many objects must clear the heap scoreboard to show the absence of a conflict for this heap root. The `ObjectList` contains a list of heap scoreboard bins that contain objects for this heap root that must be removed when the task retires. If the heap scoreboard clears the heap root, it zeros the `QueueDependences` count. Whichever approach zeros the `QueueDependences` count first for a heap root decrements the task's `NumUnresolvedDependences` count. Locks are never acquired to update these counters as the compiler generates atomic instructions to perform the updates.

If the static effects queue determines the absence of a conflict, it is possible for a task to retire before the task's heap examiners finish. When the task retires, its `DoneExecuting` flag is set. When this flag is set, the heap examiner stops and clears the traversal flag to indicate that it has halted. The task retire procedure then removes the traversed objects for the task from the heap scoreboard. If the heap examiner has not started, the task retire procedure uses an atomic operation on the traversal status flag to prevent the heap examiners for the given task record from traversing its heap roots.

5.2 Heap Scoreboards

DOJ uses a heap scoreboard to track at the object granularity the potentially conflicting heap effects between tasks. Figure 9 illustrates the heap scoreboard data structure. The heap scoreboard is implemented as an array of *access queues*. The array is indexed by the hash of the affected object's object identifier.[1]

Each access queue keeps track of potential conflicts between tasks that access objects whose object identifiers hash to that queue. Access queues are implemented as a singly linked list. The heap effects of newly dispatched tasks are enqueued at the tail. Once a heap effect for a task t reaches the head of an access queue, the given effect does not conflict with any task that was dispatched before task t. We say that such effects are *resolved*. When a task retires, its heap effects are removed from the access queues.

[1] We use object identifiers instead of object pointers to support garbage collection without having to rebuild the heap scoreboard.

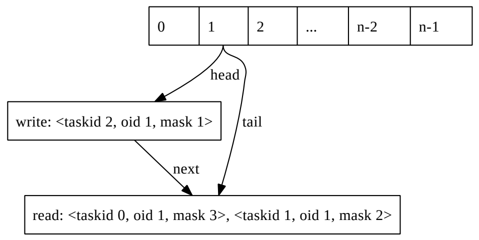
0
1
2
...
n-2
n-1
head
tail
write: <taskid 2, oid 1, mask 1>
next
read: <taskid 0, oid 1, mask 3>, <taskid 1, oid 1, mask 2>

Figure 9. Heap Scoreboard

Access queues contain two types of nodes: *read nodes* and *write nodes*. Read nodes track read effects — read effects do not conflict with each other. Therefore all consecutive read effects resolve when as a group they are at the head of the access queue. Write nodes track write effects — write effects conflict with each other and read effects. Therefore a write effect can resolve only when it reaches the head of its access queue. Each node stores both the task and a mask bitmap of the task heap roots for which that item tracks conflicts. It is important to note that a heap examiner inserts only objects with potentially conflicting effects into the heap scoreboard.

While only one heap examiner thread inserts objects into a given heap scoreboard, multiple threads can remove objects or resolve heap roots. The heap scoreboard uses atomic exchange operations on the `head` fields on the bins to lock a bin.

We typically size heap scoreboards to have a hundred thousand or more bins. False conflicts can occur if two objects hash to the same bin and in theory reduce the available parallelism. In practice, our heap scoreboards contain enough bins to prevent such conflicts from limiting parallelism in most applications.

We note that a task can retire before it clears either the heap scoreboard or static effects queues. Both data structures have been designed to allow entries to retire before they reach the head.

5.3 Static Effects Queue

DOJ uses a static effects queue to quickly track potentially conflicting accesses between tasks. DOJ generates a *conflict graph* from the results of the static analysis described in Section 4.2. A conflict graph has a node for each heap root of each task; there is an edge between two nodes if the corresponding code blocks may have a conflicting access through the corresponding heap roots. DOJ compiles the conflict graph into static effects queues that dynamically track the conflicts. The mapping problem can be viewed as a graph covering problem—to enforce the data dependence constraints, all edges in a conflict graph must be covered. The algorithm uses a greedy algorithm to try to minimize the number of queues used to cover the edges in the conflict graph.

A static effects queue has two types of queue items: a sequential item and a parallel item. A sequential item holds one task and that task must wait until the previous queue item retires and must retire before the next queue item begins. A parallel item holds multiple tasks that can execute in parallel. Each item stores both the task and a bitmap of the heap roots for which the item tracks conflicts.

6. Evaluation

We have implemented DOJ 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 4.1.2. We enabled the highest optimization level in GCC and classic optimizations in our compiler. Our implementation and benchmarks are available on the web.

We selected a diverse set of benchmarks to provide an interesting cross-section of application behaviors and a variety of al-

		Compilation Times			Speedup	
Benchmark	Lines	Sequential	OoOJava	DOJ	OoOJava	DOJ
Voronoi	4,007	3.41s	>1hr†	13.26s	N/A†	10.27×
BarnesHut	3,771	2.98s	N/A†	3.67s	N/A†	11.26×
RayTracer	3,683	1.99s	52.69s	4.18s	18.52×	18.07×
Tracking	5,419	4.31s	851.45s	6.83s	19.83×	20.05×
Crypt	2,765	1.91s	4.57s	2.63s	18.88×	19.47×
MonteCarlo	6,281	1.85s	14.45s	2.90s	22.08×	21.16×
KMeans	3,868	2.05s	13.53s	3.29s	12.60×	11.46×
MolDyn	2,565	2.04s	20.42s	4.09s	14.10×	13.91×
Power	2,565	1.94s	15.47s	2.58s	19.99×	5.58×
Labyrinth (a)	4,923	2.35s	171.52s	5.09s	11.31×	0.70×
Labyrinth (b)	4,923	2.35s	171.52s	5.09s	10.43×	9.99×
SOR	3,028	1.78s	4.65s	3.25s	9.88×	12.90×
MergeSort	2,610	1.67s	3.02s	2.14s	12.50×	10.65×

Figure 10. Compilation Times (Smaller is Better) and Speedups(Higher is Better)‡

gorithmic structures and ported them to DOJ. We included all of the large benchmarks from the Java Grande Benchmark suite [27], namely RayTracer, MolDyn, MonteCarlo, SOR, and Crypt. We selected Power and Voronoi from the JOlden benchmark suite [5], MergeSort from DPJ suite [4], Tracking from SD-VBS [30], and BarnesHut from Lonestar [19]. We selected both KMeans and Labyrinth from the STAMP benchmark suite [6] to explore benchmarks with irregular parallelism. STAMP categorizes benchmarks by metrics — KMeans and Labyrinth were categorized at opposite points on all metrics. We compiled and executed three versions of the benchmarks: *sequential* is the original sequential version, *DOJ* is the DOJ version, and *OoOJava* is the OoOJava version.

6.1 Compilation Times

A major advantage of DOJ over previous work on OoOJava is compilation time. We compiled each benchmark on a 2.27GHz 8-core Intel Nehalem Xeon with 12GB of memory. Figure 10 presents the time taken to compile each benchmark into C code. DOJ took on average 4.49 seconds to compile the benchmarks and at most 13.26 seconds. This is on average 31.15× faster than OoOJava and at most 124.69× faster than OoOJava.

6.2 Performance

We next present performance data derived from execution times averaged over 10 runs. For each experiment we computed the standard error as a percentage of the benchmark's average total execution time. The average standard error across our benchmark suite was 1.8%, with a 6.1% maximum error.

Figures 10 and 11 present speedups of the benchmark implementations generated by OoOJava and DOJ over the original sequential versions (without tasks) compiled with the same compiler. DOJ dedicates one or more cores to execute heap examiners while the other cores execute tasks, placing an upper bound of 23× on the observed speedups ignoring improvements due to cache effects. OoOJava does not dedicate any cores to a runtime component, so the upper bound for OoOJava speedups is 24×. With that in mind, the speedups observed for DOJ implementations are almost as good as—and in some cases better than—the OoOJava implementations.

Figure 12 presents the speedup curves as a function of the number of cores for the DOJ versions. We have omitted results for benchmarks that use a divide and conquer structure as they do naturally run on arbitrary numbers of cores. We observe that most benchmarks scale well to the number of cores in our test system. Note that these benchmarks use a flat task structure — restructuring the benchmarks to use hierarchically structured tasks would of course enable them to scale to much larger core counts.

Voronoi has a divide-and-conquer algorithm with a sequential merge that limits parallelism. OoOJava's compilation of Voronoi did not terminate within an hour. However, DOJ generated a parallel Voronoi implementation that achieved a significant speedup.

OoOJava cannot generate a meaningful parallel implementation of BarnesHut because all bodies are reachable from each other and limitations of the approach prevent it from determining that updates to bodies are localized. In contrast, DOJ can generate heap examiners from static effects that allow many non-conflicting updates to proceed in parallel on the single oct-tree that models the computation, which achieved a significant speedup.

DOJ achieved nearly identical performance to OoOJava for the RayTracer, Tracking, and Crypt benchmarks.

MonteCarlo has a highly parallel workload of independent simulations and a sequential task that aggregates the results. The sequential task's heap examiner checks that both the parallel task of the same iteration and the sequential task from the previous iteration have retired.

Parallelism in KMeans is limited by a non-trivial serial computation following each parallel iteration of the clustering algorithm. The serial computation is executed in a separate task which, by virtue of having three heap roots leading to possible effects, requires three heap examiners. In this case, the conflicts are between consecutive instances of the sequential task and with heap accesses made by the parent thread following the last iteration. The heap examiner efficiency is sufficient to extract the parallelism in KMeans.

Parallelism in MolDyn is similar to KMeans; however, the parallel task instances in MolDyn read from many particle objects that conflict with the particle updates in the sequential task. In this case the dynamic analysis visits many objects yet still is able to extract much parallelism.

We parallelized Power by partitioning each sub-tree of the power simulation into a task. Power is a challenging benchmark for DOJ because it performs a lightweight computation that modifies a very large number of objects. The heap examiners generated by DOJ traverses 49,140 objects per parallel phase, 26,040 of which are updated and therefore potentially conflicting. Even with such a large set of potentially conflicting updates and a relatively lightweight computation, DOJ still achieved a significant speedup.

Labyrinth is an interesting case of a speculative algorithm that can be successfully parallelized with a deterministic system. Our first experiment resulted in an execution time worse than the sequential version, shown as Labyrinth (a) in Figure 10. Each parallel task is assigned a copy of the three-dimensional array that represents the maze and calculates routes that may conflict with other routes proposed in parallel. The grid dimensions were originally specified in the order 512×512×7. A set of scratch grids is recycled among parallel tasks and in order to check that components of the scratch grid are not shared with another parallel task, a heap examiner must traverse all of the 262,144 (512×512) one dimension component array objects. In the worst case DOJ is designed to run only marginally slower than a sequential execution, because static effect queues will resolve conflicts ahead of heap examiners and allow execution of all tasks in the original sequential order. The slowdown from Labyrinth is not caused by DOJ but from Labyrinth's speculative approach to parallelism that discards conflicting routes. We verified this hypothesis by reducing the parallel

†OoOJava is overly conservative for BarnesHut and cannot generate a meaningful parallel implementation. For Voronoi, OoOJava did not complete its heap analysis within the hour we allocated.

‡All speedups are reported relative to sequential, statically compiled Java code with no tasks.

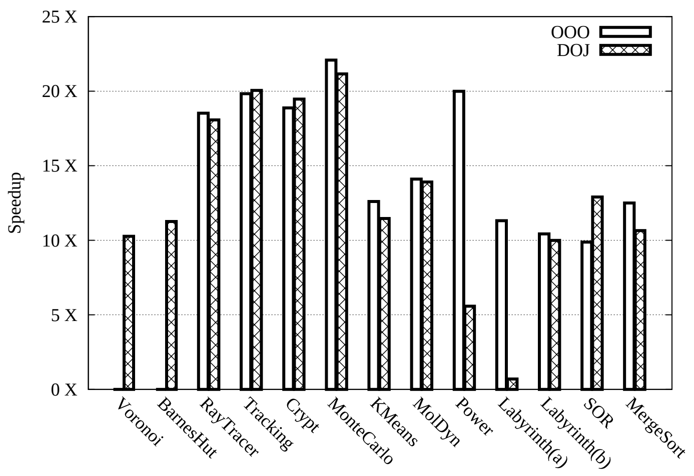

Figure 11. Benchmark Speedups Plot (Higher is Better)‡

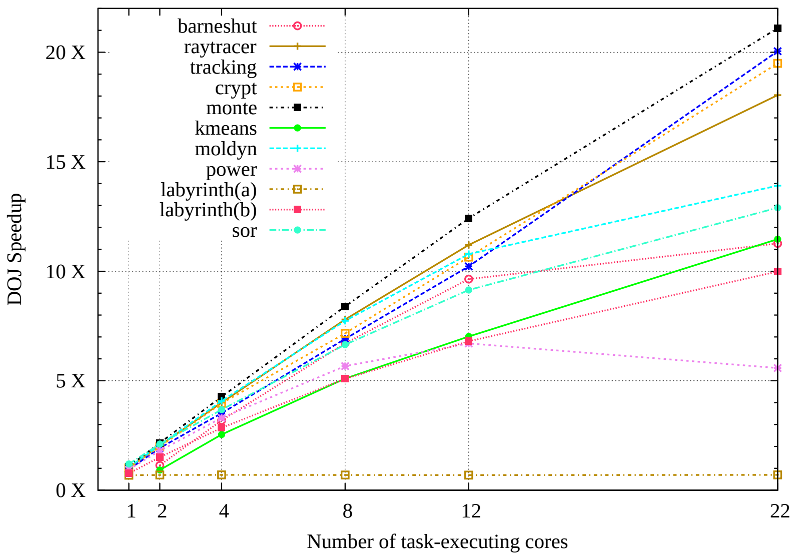

Figure 12. DOJ Speedups Scaling (Higher is Better)‡

batch size to one, which results in an execution that takes approximately the same time as the sequential version because no solutions are discarded. When we transformed the input to an equivalent problem with reordered dimensions, namely 7×512×512, the heap examiners must still visit 3,584 objects but are able to resolve conflicts quickly enough to achieve a significant speedup, reported as Labyrinth (b). While it is straightforward to handle multidimensional arrays as a special case, Labyrinth provided an interesting example of the limits of DOJ.

The speedup for SOR is limited because the benchmark is memory bandwidth-limited. Tasks access a large amount of data relative to the computation requiring significant memory bandwidth. We confirmed this by measuring the average execution of the parallel tasks for a 23-worker implementation and a 2-worker implementation of SOR. The body of parallel tasks for this benchmark performs no synchronization or system calls, so differences in execution time can only be attributed to the memory system. For 23-workers the average execution of the parallel task is 329,189 processor cycles and for 2 workers, 222,429 processor cycles.

DOJ achieved a better speedup for SOR than OoOJava because of differences in the runtime components. The generated heap examiners in the DOJ version perform traversals with no depth, and so perform a simple dynamic check similar to the check in OoOJava. However, OoOJava's runtime conflict queues must support a wider variety of ordering constraints than DOJ. As a result, DOJ can be more heavily optimized and for SOR outperforms OoOJava.

MergeSort from DPJ has a sequential merge phase that limits parallelism at all levels of recursion. However, DOJ still achieved a significant speedup.

6.3 Implementation Overheads

We measured task dispatch overhead for a microbenchmark that issues 500,000 lightweight tasks. The average time over ten executions to dispatch a task was 1886 processor cycles on the AMD Magny-Cour Opteron. For comparison, we conducted this experiment on an 8-core Intel Nehalem Xeon and measured the average of ten executions to be 982 processor cycles.

To quantify the overhead of our 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 in 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.

6.4 Parallelization Discussion

Our parallelization efforts typically began by profiling a benchmark and selecting computationally expensive blocks to enclose in tasks. We found task annotations easy to use—an inexperienced undergraduate ported the first version of Labyrinth. On average we changed only 20 lines of each benchmark.

Several of the benchmarks had a top-level loop to distribute work, making the placement of tasks easy. For other benchmarks our typical process for inserting tasks started with identifying a sequence of computationally intensive code that we expected (1) would be often repeated and (2) had no dependences between instances; we wrapped such a block in a task. Next we looked for statements dependent on the output of the parallel task along the execution path back to the definition of the parallel task. Many loops in our benchmarks worth parallelizing had parallel work at the beginning of the loop body followed by a sequence of iteration-interdependent statements in the rest of the body. Our goal was to enclose in a task any statements that would stall the parent thread from issuing new parallel task instances. Benchmarks like RayTracer and Monte had partially parallel loops that were easily partitioned into one parallel task and one sequential task.

7. Related Work

Several approaches to parallelism rely on correct developer annotations including OpenMP [7], Cilk [22], and JCilk [9]. Annotation errors in such systems can cause data races. A major advantage of DOJ is that annotation errors never cause correctness concerns.

Functional languages [20] provide strong correctness guarantees for parallel programs by prohibiting state mutation. Macro-dataflow languages [8, 14, 33] leverage the dataflow approach on larger granularity code segments. DOJ can be viewed as a system for executing sequential code using a dataflow execution model.

Speculative programming models [2, 10, 31, 32] offer a similar programming model as DOJ but can incur significant overheads to support rollback—possibly even repeated rollbacks—in the case of mis-speculation. DOJ avoids the rollback overhead by verifying that a task can safely run to completion before starting the task.

OoOJava [16, 17] uses the same programming model as DOJ. OoOJava uses sophisticated static disjoint reachability analysis [18, 21] to parallelize programs. DOJ borrows variable dependence analysis from OoOJava and significantly extends OoOJava's effects analysis to extract sufficient information to implement heap examiners. DOJ removes the need for a static reachability analysis and therefore can scale to larger code bases. DOJ's dynamic strategy relies less on the precision of static analysis and can determine the absence of conflicts even when paths exist between the conflicting objects. Moreover, DOJ can determine that tasks affect only a localized part of a data structure, while OoOJava assumes that a task can affect any reachable part of the data structure.

Synchronization Via Scheduling [3] uses a similar task-based model to DOJ but cannot analyze heap structures. They introduce a new language CDML that is similar to C++ but includes task-related constructs. They use Bloom filters to detect variable conflicts. DOJ takes a significantly more sophisticated approach to variable conflicts — our approach eliminates antidependences that would otherwise limit parallelism.

Hybrid analysis [25] shares inspiration with DOJ; both use static analysis to classify code of interest as possibly or always parallelizable, and for statically possible cases a customized dynamic analysis further refines the decision. However, whereas hybrid analysis targets Fortan arrays, calculating when code segments might access common array elements, DOJ instead targets Java-like languages which commonly layer object-oriented abstractions and where parallelism is often determined by object sharing patterns. Additionally, hybrid analysis computes a slice to calculate statically unknown addresses, and runs the slice to decide parallel safety. In the object-oriented domain a slice may contain heap modifications, so a slicing strategy will almost certainly require rollback mechanisms. DOJ avoids speculation with read-only heap examiners, which are sound approximations of code segments.

Other approaches require extensive developer annotations to avoid unchecked access to data structures [4, 24] or additional code to create serialization sets [1]. DOJ requires minimal annotations, which will likely improve developer productivity.

StarSs dynamically schedules function invocation when a function's operands are available [11]. StarSs does not analyze heap dependences and therefore restricts functions to pass-by-value and forbids passing data structures that contain pointers.

DOJ differs from inspector-executor approaches [23, 26] in that it supports complex object-oriented data structures, uses the results of static analysis to avoid inspecting most memory accesses, and does not require a runtime preprocessing phase. Van der Spek, Holm and Wijshoff describe an approach to transform pointer-traversing loops into array-traversing loops in order to apply well known array parallelization techniques [29]; the array version of a function may only be invoked when it has been previously changed and the root of its data dependencies have not changed. DOJ always attempts to execute tasks out of order and may succeed in cases where this inspector-executor approach cannot.

Decoupled software pipelining (DSWP) [15] 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 core can write to a statically identified heap region. In contrast, DOJ can execute instances of write instructions across many cores. DOJ 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 DOJ; the techniques are likely synergistic.

8. Conclusion

For parallel programming to become mainstream, parallel programming must become easier. We presented a new approach to parallel programming that uses lightweight annotations to suggest parallelization of a sequential program. DOJ generates dynamic heap dependence analyses that automatically extract heap dependences and can guarantee that the parallel execution has the same behavior as the sequential execution. We successfully parallelized twelve applications and achieved significant speedups. Moreover, we found that parallelizing applications with DOJ was straightforward and required only minor modifications to our benchmarks.

Acknowledgments

This research was supported by the National Science Foundation under grants CCF-0846195 and CCF-0725350. We would like to thank the anonymous reviewers for their helpful comments.

References

[1] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: A dynamic dependence-based parallel execution model. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 85–96, 2009.

[2] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *Proceeding of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 81–96, 2009.

[3] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword. Synchronization via scheduling: Managing shared state in video games. In *Second USENIX Workshop on Hot Topics in Parallelism*, 2010.

[4] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *Proceeding of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2009.

[5] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[6] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, 2008.

[7] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computing in Science and Engineering*, 5(1):46–55, 1998.

[8] K. Dai. Code parallelization for the LGDG large-grain dataflow computation. In *Proceedings of the Joint International Conference on Vector and Parallel Processing*, volume 457, pages 243–252. Springer Berlin / Heidelberg, 1990.

[9] J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson. The JCilk language for multithreaded computing. In *Synchronization and Concurrency in Object-Oriented Languages*, 2005.

[10] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 223–234, 2007.

[11] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero. Task superscalar: An out-of-order task pipeline. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[12] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. Reiter, H. Larkhoon, L. Ji, Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2006.

[13] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2011.

[14] C. Huang and L. V. Kale. Charisma: Orchestrating migratable parallel objects. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing*, pages 75–84, 2007.

[15] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 121–130, 2010.

[16] J. C. Jenista, Y. Eom, and B. Demsky. OoOJava: An out-of-order approach to parallel programming. In *Second USENIX Workshop on Hot Topics in Parallelism*, 2010.

[17] J. C. Jenista, Y. Eom, and B. Demsky. OoOJava: Software out-of-order execution. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2011.

[18] J. C. Jenista, Y. Eom, and B. Demsky. Using disjoint reachability for parallelization. In *Proceedings of the 20th International Conference on Compiler Construction*, 2011.

[19] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[20] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, A. J. Rebón, and P. W. Trinder. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003.

[21] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Proceedings of the Symposium on Principles of Programming Languages*, 2007.

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

[23] L. Rauchwerger, N. M. Amato, and D. A. Padua. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th International Conference on Supercomputing*, pages 137–146, 1995.

[24] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26:28–38, 1993.

[25] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: Static & dynamic memory reference analysis. *International Journal on Parallel Programming*, 31:251–283, 2003.

[26] J. H. Saltz and R. Mirchandaney. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[27] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel Java Grande benchmark suite. In *Proceedings of the SC2001*, 2001.

[28] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[29] H. L. A. van der Spek, C. W. M. Holm, and H. A. G. Wijshoff. How to unleash array optimizations on code using recursive data structures. In *Proceedings of the 24th International Conference on Supercomputing*, pages 275–284, 2010.

[30] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *Proceedings of the IEEE International Symposium on Workload Characterization*, 2009.

[31] C. von Praun, L. Ceze, and C. Caşcaval. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–89, 2007.

[32] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proceeding of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 40, pages 439–453, 2005.

[33] J. Zhou and B. Demsky. Bamboo: A data-centric, object-oriented approach to multi-core software. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.