OoOJava: An Out-of-Order Approach to Parallel Programming

James C. Jenista Yonghun Eom Brian Demsky

Abstract

Developing parallel software using current tools can be challenging. Developers must reason carefully about the use of locks to avoid both race conditions and deadlocks. We present a compiler-assisted approach to parallel programming inspired by out-of-order hardware. In our approach, the developer annotates code blocks as reorderable to decouple these blocks from the parent thread of execution. OoOJava uses static analysis to extract all data dependences from both variables and data structures to generate an executable that is guaranteed to preserve the behavior of the original sequential code.

We have implemented OoOJava and achieved significant speedups for a ray tracer and a K-Means cluster benchmark. The straightforward development model, compiler feedback, and speedups are promising indicators that a simple deterministic parallel programming model with strong guarantees can become mainstream.

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. These tools require developers to expose parallelism as threads and then control concurrent access to data with locks. Experience shows that applications written in this model are prone to both race-conditions and deadlocks.

Deterministic parallel programming models greatly simplify developing parallel code[2]. These models eliminate the race conditions that make parallel programming difficult. Several systems for deterministic parallel programming exist today. These systems have the same goal: 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) [4], constrains the usage of data structures [8, 10], requires extensive annotations [13, 14], or only guarantees determinism under unchecked conditions (disjoint data structures) [18, 5, 11].

Hardware has long used out-of-order execution to extract unstructured parallelism from sequential instruction streams [19]. Processors dynamically extract instruction dependences and then execute the instructions out-of-order while preserving the sequential stream's dependences. This paper introduces OoOJava, a parallel programming approach based on out-of-order execution.

OoOJava is a compiler-based approach that leverages both annotations and static analysis to provide a deterministic parallel programming model. OoOJava extends sequential Java with a single annotation, which developers use to indicate that a code block should be consider for out-of-order execution. OoOJava executes such blocks as soon as their data dependences are resolved. OoOJava guarantees that the execution of an annotated program respects all data dependences in the *serial elision*, the sequential program obtained by removing all annotations. Therefore annotations do not affect the program's correctness, but merely its performance. OoOJava presents minimal overheads to developers and makes guarantees that simplify parallel programming.

This basic approach has been known for some time. We add a new kind of dependence analysis, disjoint reachability analysis [6], to augment our modified approach to effects [7]. OoOJava uses the results of disjoint reachability analysis to generate a handful of lightweight comparisons that allow it to safely extract parallelism even when the heap accesses cannot be statically determined to be disjoint. OoOJava differs from inspector-executor approaches [12, 16] 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. We combine this with a new value forwarding approach that is analogous to register renaming and eliminates write-after-write and write-after-read hazards. Together, these techniques allow OoOJava to parallelize a wider-range of programs and require fewer changes to sequential code than previous systems.

This paper makes the following contributions:

- **OoOJava:** It presents a deterministic parallel programming model that extends Java with a single annotation while preserving the program's sequential semantics. Because OoOJava makes no major changes to Java, developers will likely find it easy to use.
- **Dependence Analysis:** OoOJava uses static analysis to discover data dependences.
- **Software-Based Out-of-Order Execution:** Microprocessors leverage out-of-order execution to extract fine-grained, unstructured parallelism in the instruction stream. OoOJava adapts out-of-order execution techniques in software to parallelize blocks of Java code and guarantees that the execution respects all dependences.

- **An Implementation and Evaluation:** We have implemented OoOJava and have evaluated its performance on two benchmarks.

The remainder of the paper is organized as follows. Section 2 presents the programming model using an example. Section 3 presents the execution model and Section 4 presents the static analysis components. Section 5 presents our evaluation; we conclude in Section 6.

2 Programming Model

Figure 1 presents a graph computation that will serve as an example for the OoOJava programming model. It takes a source as input and modifies a unique data structure at every node reachable from the source. The return value is an aggregation of node-local contributions.

```
1  public double findGraphTotal(Node source) {
2    double t=0;
3    Set<Node> discovered=new HashSet<Node>();
4    Queue<Node> toVisit=new Queue<Node>();
5    toVisit.push(source);
6    discovered.add(source);
7    while(!toVisit.isEmpty()) {
8      Node u=toVisit.pop();
9      for(Iterator<Node> i=n.neighbors();
10         i.hasNext(); ) {
11       Node v=i.next();
12       if(!discovered.contains(v)) {
13         toVisit.push(v);
14         discovered.add(v);
15       }
16     }
17     Data d=u.data;
18     rblock p {
19       double c=ModifyAndCompute(d);
20     }
21     rblock s {
22       t=t+c;
23     }
24   }
25   return t;
26 }
```

Figure 1: An example method that operates on a graph

Consider this method without the `rblock` keyword or associated curly braces. Line 19 of Figure 1 is dependent on the value of `d`, a unique object for every loop iteration. The summing operation in line 22 depends on the current values of `t` and `c`. Many auto-parallelizing systems would have difficulty running iterations of this loop concurrently because they cannot determine whether memory accesses in line 19 are independent.

The OoOJava parallel programming model extends the sequential programming model with reorderable blocks or *rblocks*, that decouple the enclosed code from the parent thread of execution. We declare a reorderable block with the keyword `rblock` and an identifier followed by an open curly brace, statements, and a close brace. Note that the identifier has no meaning for execution; it is used by the compiler to communicate back to the developer. The notation is similar to other code blocks except that reorderable blocks do not introduce a new variable scope and must have a single exit.

In Figure 1 reorderable block `p` is declared around line 19, which is parallelizable if `d` is unique for each iteration and the objects reachable from different instances of `d` are disjoint. A separate reorderable block `s` is declared around line 22, which must be serialized.

OoOJava addresses parallelism opportunities that can be difficult for similar systems to take advantage of. For example, CellSs [10] and Cilk [11] require explicit synchronization before accessing the results of out-of-order computations. CellSs prohibits heap references in out-of-order computations. Cilk allows heap references but without careful attention can generate incorrect results. Cilk could perform the aggregation in reorderable block `s` of this example with *inlets*, however inlets do not necessarily execute in the same order as prescribed by the serial elision which would break a non-commuting operation. Moreover, Cilk cannot schedule an arbitrary chain of out-of-order computations, while OoOJava can.

The guarantees for CellSs, Cilk, and similar parallel programming models can be broken by developer annotation mistakes, unlike OoOJava. In the worst case, a OoOJava program will simply result in a sequential execution of reorderable blocks.

3 Execution Model

OoOJava's execution model is inspired by out-of-order processors [19]. An out-of-order processor receives as input an instruction stream. When the processor *issues* the next instruction in the stream it notes any dependences of the instruction on previously issued instructions. When an instruction's dependences are resolved and the necessary functional unit is available that instruction is *dispatched* to the functional unit. A completed instruction *retires* by updating the processor state.

A parent reorderable block issues a child reorderable block by allocating a record for it and making a runtime count of any outstanding dependences the child has on siblings. A parent executes its own code until reaching a child reorderable block, where it then issues the child and skips to the end of that child's definition to continue executing. When all dependences for an issued reorderable block are resolved it is dispatched. Once a reorderable block reaches its single-exit it must wait for all of its children to retire before it retires. Similar to register renaming in out-of-order processors, OoOJava is able to avoid write-after-write and write-after-read hazards by having a reorderable block forward values directly to a dependent reorderable block.

Reorderable blocks can be nested and there is an implicit top-level reorderable block for the main method; an important property of OoOJava is that reorderable blocks form a tree at runtime. A parent is responsible for issuing children and managing the dependences among its children and itself, therefore reorderable blocks in

different subtrees can proceed concurrently on different processing cores and parallelize the scheduling load.

In the same fashion that out-of-order processors can be viewed as a restricted form of dataflow computation, OoOJava can be viewed as dynamically translating imperative programs into dataflow computations.

4 Dependence Analysis

OoOJava guarantees that parallelized programs preserve the behavior of their serial elisions. To make this guarantee, OoOJava must ensure that parallelized program executions preserve all data dependences between reorderable blocks. Reorderable blocks form a hierarchy in the program. Preserving parent-child dependences and sibling dependences suffices to preserve all dependences.

OoOJava uses static analysis to conservatively extract data dependences and then uses the analysis results to generate runtime checks that preserve the dependences. These data dependences fall into two categories: variable dependences and heap dependences.

4.1 Variable Dependence Analysis

The variable dependence analysis ensures that each reorderable block reads the correct values from its variables. OoOJava performs the variable dependence analysis on each reorderable block that can serve as a parent to extract variable dependences between that reorderable block and its children. The variable dependence analysis begins by computing each reorderable block's *in-set*, the set of variables a reorderable block requires values from before being dispatched, and each reorderable block's *out-set*, the set of variables a reorderable block may write new values to that either a parent or sibling reorderable block may read. In the example, reorderable block `p` has in-set $\{\texttt{d}\}$ and out-set $\{\texttt{c}\}$ while reorderable block `s` has in-set $\{\texttt{t}, \texttt{c}\}$ and out-set $\{\texttt{t}\}$.

For each program point and every live variable at that program point, the analysis uses a fixed-point algorithm to compute the source of that variable's value. The *variable source* for a variable v has the form $\langle r, a, v_o \rangle$. The combination of r and a statically specify a dynamic instance of a reorderable block: r is a reorderable block and a is the age of that reorderable block (i.e. how many instances of that reorderable block have been issued between the source instance and the current program point). To bound the analysis, a is taken from $\{0, 1, \top\}$, where a variable source with $a = \top$ means an instance with an unknown age. The variable v_o specifies which variable in the out-set of the reorderable block given by r and a contains the relevant value.

The variable dependence analysis results at line 19 of Figure 1 would show that variable `d` has only one possible variable source: $\langle \text{parent}, 0, \texttt{d} \rangle$. Therefore `d`, in `p`'s in-set, is classified as *ready*, indicating that its value is always available when `p` is issued.

Reorderable block `s` has in-set $\{\texttt{c}, \texttt{t}\}$, where `c` has the variable source $\langle \texttt{p}, 0, \texttt{c} \rangle$, the most recent instance of `p`. Therefore `c`, in `s`'s in-set, is classified as *static*, indicating that the variable's source comes from exactly one statically-named reorderable block and therefore the code does not need to dynamically track the source of `c`.

When a parent issues a child, it has enough information to inform the sibling that serves as a source reorderable block that it should forward the value of the variable when it retires to the child currently being issued. If the sibling has already retired, the parent forwards the value immediately and marks the dependence resolved. When a reorderable block retires it has a list of issued reorderable blocks that are waiting for the values it computed.

The third classification for variable sources is *dynamic*, such as in-set variable `t` for reorderable block `s`. Dynamic means that the exact reorderable block instance that produces the value is not known statically. In the example, the value for `t` will either come from the parent in line 2 or from the previous instance of `s`. The analysis would compute that `t`'s possible sources are $\langle \text{parent}, 0, \texttt{t} \rangle$ and $\langle \texttt{s}, 0, \texttt{t} \rangle$. If an in-set variable is classified as dynamic then code is generated at control-flow joins backwards from the reorderable block to dynamically track the source reorderable block instance.

Whenever a parent statement has a dependence on a child reorderable block, the parent stalls until that child retires. The exception to this rule is the copy statement as discussed in Section 4.1.1.

4.1.1 Source Copying

When a statement of a parent reorderable block depends on a value that is produced by its child, the parent must stall until the child retires. Line 25 of the example is an instance of this case where the parent must stall for the final instance of `s` to compute the value of `t`. When a parent stalls on a child for a statically resolved dependence, the parent copies all variables from that child with statically resolved dependences.

Consider the following code fragment:

```
1 rblock child1 {
2   y=1;
3 }
4 x=y;
5 rblock child2 {
6   print(x);
7 }
```

In an effort to minimize stalls, we allow variable sources to be copied by the analysis at a copy statement and then omit the statement during code generation. If the variable `y` has the source $\langle \texttt{child1}, 0, \texttt{y} \rangle$ before the copy statement in line 4, the analysis would compute `x`'s source after the statement as $\langle \texttt{child1}, 0, \texttt{y} \rangle$ also. In the runtime implementation `parent` would issue `child1`, skip the copy statement, and then issue `child2` with the unresolved dependence that variable `x` for `child2` should be obtained from `y` in `child1`'s out-set.

4.1.2 Virtual Reads

Dependences can be affected by the internal control flow of a reorderable block. If a variable is conditionally modified by a reorderable block, we say that it is virtually read. By conservatively including these variables in the in-sets we simplify dynamic dependence tracking. This ensures that the sources for variables are known when a reorderable block is issued.

4.2 Heap Dependence Analysis

Aliasing is known to make automatically parallelizing code that modifies data structures difficult. Previous approaches have simplified the problem. Jade [13] and DPJ [14] allow operations on data structures but require additional annotations to characterize memory accesses.

OoOJava leverages a new static analysis, disjoint reachability analysis [6], that extracts static heap reachability properties. OoOJava uses these properties to generate lightweight dynamic checks that ensure the execution of reorderable blocks respect the memory dependences of the program's serial elision.

OoOJava reasons about memory accesses in terms of *heap roots*, a root 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 reorderable block or the first object along a heap path accessed by a parent reorderable block after issuing a child.

Two reorderable blocks only have a heap dependence if both of the following two conditions are true. The first necessary condition is that one reorderable block writes to a field f of an object allocated at site s and the second reorderable block either reads or writes to the same field of an object allocated at the same site. We call such an access a potentially conflicting access and an object allocated at this site a potentially conflicting object. The second necessary condition is that there must exist a potentially conflicting object that is reachable from the heap roots of both reorderable blocks that were used to perform the potentially conflicting access.

4.2.1 Effects Analysis

OoOJava uses heap effects to abstract the read and write heap operations a reorderable block performs. The effects analysis computes a conservative set of heap effects for child reorderable blocks and the sections of code in the parent reorderable block between subsequent child reorderable blocks. Effect are associated with the heap roots used to obtain a reference to the affected object.

Heap effects have the form $\langle r, o, s, f \rangle$, where r is the allocation site of a heap root that was used to obtain a reference to the affected object, o is a read, write, or reachability change effect, s is an allocation site, and f is a field (or a special symbol for array elements). Note that it is possible for a heap root to have multiple allocation sites. OoOJava uses an interprocedural, backwards dataflow analysis to compute heap effects.

When the effects analysis considers line 19 of the example it will use the interprocedural results to discover that write operations are performed on `Data` objects through the heap root referenced by `u`. The effects analysis has therefore discovered a potential data dependence between instances of `p`.

4.2.2 Disjoint Reachability Analysis

If the effects analysis cannot rule out a potential conflict, disjoint reachability analysis may determine that the concrete objects represented by the heap roots cannot both reach the same potentially conflicting object.

Disjoint reachability analysis extends a standard points-to graph with reachability annotations. The standard points-to graph uses heap region nodes to abstract objects and edges to abstract references. There are two heap region nodes for each allocation site — one abstracts the most recently allocated object from the site and the other summarizes all older objects. Disjoint reachability analysis annotates these heap region nodes and edges with sets of reachability states, namely sets of heap region nodes and arities. If a heap region h_0 at some program point has the reachability state $[h_1, h_2^*]$, it means that objects represented by h_0 may be reachable from at most one object represented by h_1, any number of objects represented by h_2, and exactly zero objects from every other heap region in the points-to graph.

Note that these specific disjoint reachability properties are not computed by either current pointer analyses [1] or shape analyses [15, 9].

Reachability states capture disjoint reachability information even for objects that are represented by the same node. Consider two effects, $\langle r_0, w, s, f \rangle$ and $\langle r_1, w, s, f \rangle$ that both write to the f field of an object from allocation site s through different heap roots. OoOJava uses disjoint reachability analysis to determine whether the possible target objects of the effects can ever be the same object when accessed through the given heap roots.

There are three possible answers to this kind of query. The first is that a single object allocated at allocation site s cannot be reached from both heap roots r_0 and r_1. In this case the second condition for a memory conflict fails and there is no heap dependence.

If the heap roots r_0 and r_1 are abstracted by the same heap region h in disjoint reachability analysis and the reachability states for `s` contain h with arity 1, then the effects only conflict if the object abstracted by r_0 and the object abstracted by r_1 are the same. In this case there is a fine-grained conflict and generated code does a simple dynamic comparison: if r_0 and r_1 are different objects at runtime then the effects do not conflict.

In the remaining case, the analysis cannot eliminate a

possible conflict. In this case, we say that the two heap roots have a coarse-grained conflict and the corresponding code blocks must be executed sequentially.

In general, object reachability at different program points cannot be compared. However, as long as heap operations only increase reachability of an object, the reachability states are comparable. If an operation eliminates an edge (performs a strong update in the analysis) and therefore may decrease reachability, we associate a reachability decrease effect with that operation. Reachability decrease effects conflict with all other effects on objects that are reachable from the same heap roots.

4.2.3 Conflict Graphs

OoOJava generates a conflict graph from the results of the effects analysis and the disjoint reachability analysis. There is a conflict graph for each parent reorderable block that captures the heap dependences between the parent reorderable block and its child reorderable blocks. There is a node in the graph for each heap root. There are two types of edges in the graph: coarse-grained edges indicate that the corresponding code blocks cannot be reordered and fine-grained edges indicate that the corresponding code blocks can only be reordered if the heap roots are different objects. The absence of an edge indicates that the code blocks have no heap conflicts. The edges are generated using the conditions described in the previous sections.

The value of a pointer may be unknown when a reorderable block is issued. Because the value is unknown, fine-grained conflicts checks must conservatively assume that it matches other potentially conflicting effects.

4.2.4 Compiling Conflict Graphs

OoOJava compiles conflict graphs into a set of runtime queues that enforce the dependence constraints. The runtime includes a queue implementation that enforces both coarse-grained and fine-grained constraints. The queue supports a number of different access modes each of which enforces a number of fine-grained or coarse-grained conflict constraints with other access modes. The queue has a corresponding conflict graph that captures the ordering constraint the queue enforces.

OoOJava maps a conflict graph to a set of runtime queues. 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 queue implementations used to cover the edges in the conflict graph.

5 Evaluation

We have implemented OoOJava and evaluated it on a 2.27GHz 8-core Intel Xeon with 12GB of memory. We report results for two benchmarks: RayTracer and KMeans. The system and benchmarks are available at `http://demsky.eecs.uci.edu/compiler.php`.

RayTracer was taken from the Java Grande benchmark suite [17]. It renders a scene at a resolution of 150×150 pixels. Averaged over 10 runs, we obtained a speedup of $6.14\times$ on 8 cores relative to the sequential Java version of RayTracer. Iterations of RayTracer's outer loop compute one row of the scene image. We used OoOJava to target the parallelism in this loop by declaring a reorderable block around the independent row computations and a reorderable block around the subsequent code that collects the row results and computes a running checksum. We compiled the benchmark and obtained results similar to a sequential execution.

Upon inspection of the conflict graph, OoOJava correctly reported that what we thought were independent reorderable blocks had fine-grained conflicts on objects referenced by the singleton `RayTracer` object. The source code revealed that RayTracer has real heap access conflicts: it allocates scratch objects once and reuses them in each row iteration. Using that information we moved the scratch object allocations into the loop, eliminating the data dependences between iterations. OoOJava was then able to obtain the reported speedup.

The KMeans benchmark groups objects in an N-dimensional space into K clusters. The algorithm is used to partition data items into related subsets. We ported it from the STAMP benchmark suite [3]. Our implementation differs from the original version in that it does not use transactions to update the shared data structures, instead a single thread identifies the clusters for several data points and then updates the cluster data structures. We split the main loop into two reorderable blocks: the first finds the nearest clusters and the second updates the cluster data structures. Averaged over 10 runs, we obtained a speedup of $5.34\times$ on 8 cores relative to the sequential Java version of KMeans.

Note that the current runtime implementation has not been fully optimized. We expect that further optimizations will improve the speedups for both benchmarks.

6 Conclusion

For parallel programming to become mainstream, parallel programming tools must become easy to use. This paper presents an approach to parallel programming that uses annotations to specify how to parallelize 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. Our initial results are promising — we have achieved significant speedups on our benchmarks. Moreover, parallelizing the benchmarks with OoOJava was straightforward.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[2] R. L. Bocchino, V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *First USENIX Workshop on Hot Topics in Parallelism*, 2009.

[3] 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*, September 2008.

[4] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[5] 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.

[6] J. C. Jenista and B. Demsky. Disjointness analysis for Java-like languages. Technical Report UCI-ISR-09-1, University of California, Irvine, 2009.

[7] P. Jouvelout and D. Gifford. The FX-87 interpreter. In *Proceedings of the 1988 International Conference on Computer Languages*, 1988.

[8] 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 Symbol. Comput.*, 16(3):203–251, 2003.

[9] M. Marron, M. Méndez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *Proceedings of Program Analysis for Software Tools and Engineering*, pages 43–49, New York, NY, USA, 2008. ACM.

[10] J. P. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it easier to program the Cell broadband engine processor. *IBM J. Res. Dev.*, pages 593–604, 2007.

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

[12] 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.

[13] 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.

[14] R.L. Bocchino et al. 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.

[15] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Transactions on Programming Languages and Systems*, 2002.

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

[17] L. A. Smith, J. M. Bull, and J. Obdrzalek. A parallel Java Grande benchmark suite. In *Supercomputing*, 2001.

[18] J. Subhlok and B. Yang. A new model for integrated nested task and data parallel programming. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1997.

[19] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11(1), 1967.