A Shape Analysis for Optimizing Parallel Graph Programs *

Dimitrios Prountzos[1] Roman Manevich[2] Keshav Pingali[1,2] Kathryn S. McKinley[1]
[1]Dept. of Computer Science, The University of Texas at Austin, Texas, USA.
[2]Institute for Computational Engineering and Sciences, The University of Texas at Austin, Texas, USA.
{dprountz@cs.utexas.edu,roman@ices.utexas.edu,pingali@cs.utexas.edu,mckinley@cs.utexas.edu }

Abstract

Computations on unstructured graphs are challenging to parallelize because dependences in the underlying algorithms are usually complex functions of runtime data values, thwarting static parallelization. One promising general-purpose parallelization strategy for these algorithms is optimistic parallelization.

This paper identifies the optimization of optimistically parallelized graph programs as a new application area, and develops the first shape analysis for addressing this problem. Our shape analysis identifies *failsafe* points in the program after which the execution is guaranteed not to abort and backup copies of modified data are not needed; additionally, the analysis can be used to eliminate redundant conflict checking. It uses two key ideas: a novel *top-down* heap abstraction that controls state space explosion, and a strategy for predicate discovery that exploits common patterns of data structure usage.

We implemented the shape analysis in TVLA, and used it to optimize benchmarks from the Lonestar suite. The optimized programs were executed on the Galois system. The analysis was successful in eliminating all costs related to rollback logging for our benchmarks. Additionally, it reduced the number of lock acquisitions by a factor ranging from $10\times$ to $50\times$, depending on the application and the number of threads. These optimizations were effective in reducing the running times of the benchmarks by factors of $2\times$ to $12\times$.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.1.3 [*Programming Techniques*]: Object-oriented Programming; F.3.2 [*Logics and Meanings of Programs*]: Program analysis

General Terms Algorithms, Languages, Performance, Verification

Keywords Abstract Interpretation, Compiler Optimization, Concurrency, Parallelism, Shape Analysis, Static Analysis, Amorphous Data-parallelism, Irregular Programs, Optimistic Parallelization, Synchronization Overheads, Cautious Operators.

* This work was supported by NSF grants 0833162, 0719966, 0702353, 0724966, and 0739601, as well as grants from the IBM and Intel Corporations.

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

POPL'11, January 26–28, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0490-0/11/01...$10.00

1. Introduction

Computations on large, unstructured graphs are ubiquitous in many problem domains such as computational biology, machine learning, and data mining. They are difficult to parallelize because most dependences between computations in these algorithms are functions of values known only at runtime such as the structure of the (possibly mutating) graph; therefore, it is impossible to parallelize these algorithms statically using techniques such as shape analysis and points-to analysis [27].

One general-purpose solution to parallelizing graph computations is optimistic parallelization: computations are performed speculatively in parallel, but the runtime system monitors conflicts between concurrent computations, and rolls back offending computations as needed. There are many implementations of this high-level idea such as thread-level speculation [31], transactional memory [9, 12], and the Galois system [17]. For concreteness, our results are presented in the context of the Galois system but they are applicable to other systems as well.

In the Galois system, applications programmers write algorithms in sequential Java augmented with a construct called the *Galois unordered-set iterator*[1]. This iterator iterates in some unspecified order over a set of *active nodes*, which are nodes in the graph where computations need to be performed. The body of the iterator is considered to be an *operator* that is applied to the active node to perform the relevant computation, known as an *activity*. An activity may touch other nodes and edges in the graph, and these are collectively known as the *neighborhood* for that activity. These nodes and edges must be accessed by invoking methods from graph classes provided in the Galois library.

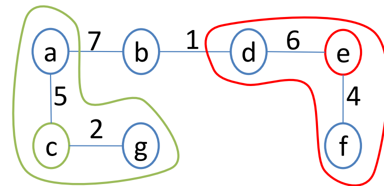

Figure 1. Neighborhoods in Boruvka's MST algorithm

We illustrate these concepts using Boruvka's minimal spanning tree (MST) algorithm [7], the running example in this paper. The MST starts as a forest with each node in its own component. The algorithm iteratively contracts the graph by *non-deterministically* choosing a graph node, examining all edges incident on that node to find the lightest weight edge, and contracting that edge, which is

[1] The Galois system also supports ordered-set iterators, but we do not consider these in this paper.

added to the MST. The algorithm terminates when the graph has been contracted to a single node. Figure 1 shows an undirected graph. For active node **e**, the neighborhood of the corresponding activity consists of nodes **d**, **e** and **f**, and the edges between these nodes, since these are the edges that must be examined to find and contract the lightest weight edge connected to **e**.

In most algorithms, each neighborhood is a small portion of the overall graph, so it is possible to work on many active nodes concurrently provided the corresponding neighborhoods do not overlap. For example, in Figure 1, the neighborhood for the activity at node **c** is disjoint from the neighborhood for the activity at node **e**, so these activities can be performed in parallel. However, the activity at node **b** cannot be performed concurrently with the activity at **e** since the neighborhoods overlap.

In the Galois system, this concurrency is exploited by adding all graph nodes to the work-set and executing iterations of the Galois set-iterator speculatively in parallel. All concurrency control is performed within the library graph classes. Conceptually, an exclusive lock called an *abstract lock* is associated with each graph element, and this lock is acquired by an activity when it touches that element by invoking a graph API method. If the lock has already been acquired by another activity, a conflict is reported to the runtime system, which rolls back offending activities. To permit rollback, methods that modify the state of the graph also record undo actions that are executed on rollback. The idea of handling conflicts at the abstract data type level rather than at the memory level is also used in boosted TM systems [11]. The Galois system has been used to parallelize complex graph algorithms in the Lonestar benchmark suite [15].

Compared to static parallelization, optimistic parallelization has several overheads.

1. *Wasted work from aborted activities*: Because conflicts between activities are detected online, an activity may be rolled back after it has performed a lot of computation.
2. *Conflict checking*: Abstract locks must be acquired and released by activities, and this is an overhead even if no activities are ever aborted.
3. *Undo actions*: These must be registered for every graph API call that might mutate the graph.

In this paper, we present a novel shape analysis that can be utilized to reduce the overheads of conflict checking and registering undo actions (reducing the number of aborted activities is mainly a scheduling problem, and is dealt with elsewhere [16]). Our main contributions are the following.

- *Shape Analysis*: We develop a novel shape analysis for programs with set and graph data structures, which infers properties for optimizing speculative parallel graph programs. We utilize the structure of stores arising in our programs to design a *hierarchy summarization abstraction*, which uses a finite set of reachability relations relative to a given property (the "object-is-locked" property), to abstract stores into shape graphs . Our abstraction assigns unary predicates *only to root objects*, capturing reachability facts from root objects to objects deeper in the heap. Thus, the size of an abstracted store is *linear in the number of variables*, and the number of abstracted stores at a program point depends on the number of explored variable-alias sets, which tends to be constant in our programs ($\approx$6). Therefore the number of abstract states explored by our analysis in practice is linear in the size of the program, circumventing the state-space explosion that is the bane of existing shape analyses.
- *Predicate Discovery*: We develop a simple yet effective technique for discovering predicates relevant for inferring the set objects that are always locked, at each program location, from data structure specifications and "footprints" of data structure method specifications.
- *Evaluation of effectiveness*: We implement our shape analysis in the TVLA framework and use a Java-to-TVLA front-end to analyze several benchmarks from the Lonestar Benchmark Suite [15], a collection of real-world graph-based applications that exhibit irregular behavior. Our analysis takes at most 16 seconds on each benchmark and infers all available optimizations. These optimizations result in substantial improvements in running time, ranging from $2\times$ to $12\times$.

Several existing heap abstractions, including Canonical Abstraction [32], Boolean heaps [28], indexed predicate abstraction [20], and generalized typestates [21], abstract the heap by recording a set of unary predicates for **every object** and summarizing the heap by collapsing equivalence classes of objects with the same set of predicate values. Such abstractions achieve high precision, as they express every Boolean combination of intersection and union of objects satisfying those predicates. However, the size of a summarized heap can be exponential in the number of predicates, and the summarization of a set of stores can be doubly-exponential. We call these *bottom-up abstractions*, since they typically express reachability facts for objects in the depth of the heap relative to heap roots. Our experience with bottom-up abstraction shows that heaps are partitioned very finely, leading to state space explosion. As we discuss in Section 4, our *top-down abstraction* runs several orders of magnitude faster than an implementation of the bottom-up abstraction approach when analyzing our benchmarks.

The rest of the paper is organized as follows. Section 2 provides an overview of our optimizations and shape analysis on Boruvka's MST example. Section 3 presents our shape analysis via hierarchy summarization and predicate discovery. Section 4 describes the static analysis implementation and gives experimental results that demonstrate the effectiveness of the approach presented in this paper. Section 5 discusses related work.

2. Overview

This section introduces the programming model, the performance optimizations, and our shape analysis informally, using Boruvka's MST algorithm as the running example.

2.1 Boruvka's MST algorithm

Pseudocode for the algorithm is shown in Figure 2. The Galois iterator on line 25 iterates over the graph nodes in some non-deterministic order, performing edge contractions. In lines, 31-38 we examine the neighbors of the active node `a`, and identify the neighbor `lt`, which is connected to `a` by a lightest weight edge. In lines 44-59 we contract the two components by removing `lt` from the graph, and updating all of `lt`'s neighbors to become neighbors of `a`. This is done by the loop in lines 45-58. If a neighbor `n` of `lt` is already connected to `a`, we update the data value of the edge connecting them (lines 49-54). Otherwise, we add an edge connecting the two nodes (lines 55-57).

In Boruvka's algorithm, the neighborhood of an active node `a` consists of the immediate neighbors of `a` and `lt` and their related edges and data. In more complex examples like Delaunay mesh refinement, the neighborhood of an activity can be an unbounded subgraph.

2.2 Speculative Execution in Galois

The graph data structure `Graph<ND,ED>` is parameterized by data objects referenced by nodes and edges, respectively. The work list of active nodes is stored in a set `GSet<Node>`. The `Weight` ob-

```
1  class GaloisRuntime {
2    @rep static set<Object> locks; // abstract locks
3    // Flag options
4    static int LOCK_UNDO=0; // acquire locks + log undo
5    static int UNDO      =1; // log undo
6    static int LOCK      =2; // acquire locks
7    static int NONE      =3; // no locks and no undo
8  }
9
10 class Weight {
11   static Weight MAX_WEIGHT;
12   int v;
13   // We record the end−points of the edge that
14   // holds the weight in the input graph.
15   final Node<Void> initSrc, initDst;
16   int compareTo(Weight other);
17 }
18 class Boruvka {
19   void main() {
20     Graph<Void,Weight> g = ...// read from file
21     GSet<Node> wl = new GSet<Node>();
22     wl.addAll(g.getNodes(NONE), NONE);
23     GBag<Weight> mst = new GBag<Weight>();
24
25     // Galois iterator
26     foreach (Node a : wl) { // in any order
27 L1:    Set<Node> aNghbrs = g.getNeighbors(a, LOCK);
28       // Find neighbor incident to lightest edge
29       Weight minW = Weight.MAX_WEIGHT;
30       Node lt = null;
31 L2:    for (Node n : aNghbrs) { // Iterator nIter
32         Edge e = g.getEdge(a, n, NONE);
33         Weight w = g.getEdgeData(e, NONE);
34         if (w.compareTo(minW) < 0) {
35           minW = w;
36           lt = n;
37         }
38       }
39       if ( lt == null) // no neighbors
40         continue;
41       // Contract edge (a, lt )
42 L3:    g.getNeighbors(lt, LOCK); // avoids undo in L4
43 L4:    g.removeEdge(a, lt, NONE);
44 L5:    Set<Node> ltNghbrs = g.getNeighbors(lt, NONE);
45 L6:    for (Node n : ltNghbrs) { // Iterator nIter
46         Edge e = g.getEdge( lt, n, NONE);
47         Weight w = g.getEdgeData(e, NONE);
48         Edge an = g.getEdge(a, n, NONE);
49         if (an != null) { // merge edges
50           Weight wan = g.getEdgeData(an, NONE);
51           if (wan.compareTo(w) < 0)
52             w = wan; // use minimal weight
53 L7:         g.setEdgeData(an, w, NONE);
54         }
55         else { // new neighbor for a
56 L8:         g.addEdge(a, n, w, NONE);
57         }
58       }
59 L9:    g.removeNode(lt, NONE);
60 L10:   mst.add(minW, NONE);
61 L11:   wl.add(a, NONE); // put node back on worklist
62     } } }
```

Figure 2. Simplified implementation of Boruvka's algorithm.

jects, which record the weights of the MST edges and their endpoints (nodes) in the original graph, are stored in another collection `GBag<Weight>`, which only allows addition operations in a concurrent context. The last argument to a data structure method is a flag that tells the runtime system whether the method should attempt to acquire abstract locks and whether it should log an inverse method call. The default value `LOCK_UNDO` is always a safe choice, ensuring correctness of speculative execution.

The Galois system protects user-defined data types, such as `Weight`, using a read-write lock (allowing concurrent read operations but at most one write operation) and maintaining backup copies of such objects.

Iterations of the Galois set iterator are executed speculatively in parallel, and this execution has transactional semantics: an iteration either completes and commits, or is rolled back and retried.

Syntactic Categories

TName	Types
OFld	Pointer fields
SFld	Set fields
Field	All fields
PVar	Pointer variables
BVar	Boolean variables
SVar	Set-valued variables
Var	All variables

Data Types (EBNF)

$$
\begin{array}{rcl}
\textit{TypeDecl} & ::= & \texttt{class}\ \textit{TName}\{\textit{FieldDecl}^*\ \textit{MethodDef}^*\} \\
\textit{FieldDecl} & ::= & [\texttt{@rep}]\ [\texttt{static}]\ \textit{TName}\ \textit{OFld};\ | \\
 & & \texttt{@rep}\ [\texttt{static}]\ \textit{set}\langle \textit{TName}\rangle\ \textit{SFld}; \\
\textit{MethodDef} & ::= & \texttt{@locks}(\textit{Path}^*)\ \texttt{@op}(\textit{Stmt}^*)\ \texttt{Java-code} \\
\textit{Stmt} & ::= & \textit{Var} = \textit{Expr}\ |\ \textit{Var.Field} = \textit{Expr} \\
\textit{Expr} & ::= & \textit{Path}\ |\ \textit{Path} + \textit{Path}\ |\ \textit{Path} - \textit{Path}\ |\ \texttt{choose}(\textit{Path})\ | \\
 & & \textit{Path}\ \texttt{in}\ \textit{Path}\ |\ \textit{Path}\ \texttt{notIn}\ \textit{Path}\ | \\
 & & \texttt{isEmpty}(\textit{Path})\ |\ \texttt{new}\ \textit{TName}((\textit{Field} = \textit{Var})^*) \\
\textit{Path} & ::= & \textit{Var}.(\textit{Var} + \textit{Field}[:\textit{SVar}] + \texttt{rev}(\textit{Field})[:\textit{SVar}])^+
\end{array}
$$

Figure 3. EBNF grammar for specified data structures. The notation $[x]$ means that `x` is optional.

2.3 Data Structure Specifications

Figure 3 shows the syntax for a lightweight specification of abstract data types (for all clients), defining their abstract state, abstract locks acquired by each method, and operational semantics of each method in terms of abstract fields. The set of variables, includes method parameters, the special `ret` parameter for returning values, static variables, and temporary variables used to define the semantics of methods. The formal semantics of this language can be found in the accompanying report [30]. Our analysis operates in terms of these specifications, ignoring the internal details of library ADT's. We assume the correctness of the specifications; approaches such as [34] can be used for their verification.

Figure 4 shows a graph type built from the `Node` and `Edge` types and the parametric types `ND` and `ED`, used to store user-defined data on the graph nodes and edges. In our example, nodes do not store any data objects and thus their `nd` fields are null. Figure 5 shows a bag, a set, and an iterator type.

Specifying Abstract Data Types. The `@rep` annotations in Figure 4 define the abstract state of a data structure in terms of *set-fields* [18], i.e., fields whose values are sets of objects.

For example, the abstract state of `Graph` is given by a pair of sets — `ns` and `es` — representing the set of graph nodes and edges, respectively. [2] We represent an undirected edge by a single edge, directed arbitrarily.

`GaloisRuntime` contains a static (i.e., global) `locks` set, representing the set of abstract locks acquired by an iteration. [3]

Example 1. *Figure 6 shows an abstract store representing the input graph of Figure 1 where* `a` *references the active node* **a** *and* `lt` *references* **c** *— the node connected to* **a** *by the lightest edge, discovered on the second iteration after iterating over* **b**.

[2] The full specification includes additional first-order constraints.

[3] Galois implements a lock coarsening scheme by maintaining a single set of abstract locks, shared among all data structure instances.

```
class Graph<ND,ED> { // Undirected boosted graph
@rep set<Node> ns; // graph nodes
@rep set<Edge> es; // graph edges

@locks(n.rev(src).dst, n.rev(dst).src)
@op(nghbrs = n.rev(src).dst + n.rev(dst).src,
    ret = new Set<Node<ND>>(cont=nghbrs))
Set<Node<ND>> getNeighbors(Node<ND> n, int opt);

@locks(f.rev(src).dst.t, t.rev(src).dst.f)
@op(f.rev(src):eft.dst.t, t.rev(src):etf.dst.f,
    ret = choose(eft + etf))
Edge<ED> getEdge(Node<ND> f, Node<ND> t, int opt);

@locks(f, t, f.rev(src).dst.t, t.rev(src).dst.f)
@op(f.rev(src):eft.dst.t, t.rev(src):etf.dst.f,
    ret = (eft+etf) in es,
    ne = new Edge<ED>(src=f, dst=t, ed=d),
    es += ne)
boolean addEdge(Node<ND> f, Node<ND> t, ED d,
                int opt);

@locks(n.rev(src).dst, n.rev(dst).src)
@op(ret = n in ns, ns -= n,
    es -= n.rev(src) + n.rev(dst))
boolean removeNode(Node<ND> n, int opt);

@locks(f.rev(src).dst.t, t.rev(src).dst.f)
@op(f.rev(src):eft.dst.t, t.rev(src):etf.dst.f,
    ret = (etf + eft) in es, es -= (etf + eft))
boolean removeEdge(Node<ND> f, Node<ND> t, int opt);

@locks(n)
@op(ret = n.nd)
ND getNodeData(Node<ND> n, int opt);

@locks(e, e.src, e.dst)
@op(ret = e.ed)
ED getEdgeData(Edge<ED> e, int opt);

@locks(e, e.src, e.dst)
@op(e.ed = d)
void setEdgeData(Edge<ED> e, ED d, int opt);

}

class Node<ND> {
 ND nd; // data object
}

class Edge<ED> {
  Node src; // edge origin
  Node dst; // edge destination
  ED ed;    // data object
}
```

Figure 4. Graph specification samples.

The figure does not show objects used by the internal (concrete) representation of specified data types. Instead, it uses the (`@rep`) set fields to indicate that an object is contained in a set field of a data structure.

Filled locks denote objects in `GaloisRuntime.locks`; hollow locks denote objects for which our analysis infers that lock protection is not required.

Path Language. We use a language of access path expressions (access paths for short) to denote the set of objects that can be obtained by following variables and fields in a store: a variable (`x`) denotes the object it references; a pointer field `e.f` denotes an object obtained by traversing the field `f` forward from an object denoted by the prefix expression `e`; a set field denotes any object stored in the set stored in a given object; a reverse field, written `rev(f)` or $\overleftarrow{f}$, denotes objects obtained by traversing field `f` backwards. We formalize path expressions in Section 3.

```
class GSet<E> { // boosted set
 @rep set<E> gcont;// set contents
 @locks(e)
 @op(ret = e in gcont)
 boolean contains(E e);
 @locks(e)
 @op(ret = e notIn gcont, gcont += e)
 boolean add(E e);
}

class GBag<E> {//boosted bag for reduction operations
 @rep set<E> bcont; // bag contents
 @locks() // No locks required!
 @op(bcont += e)
 void add(E e);
 @op(ret = new Set<E>(bcont))
 Set<E> toSet(); // used only in sequential code
}

interface Set<E> { // sequential set from java.util
 @rep set<E> cont; // set contents
}

interface Iterator<E> { // iterator from java.util
 @rep Set<E> all;     // underlying set
 @rep set<E> past;    // past iteration elements
 @rep E at;           // element at current iteration
 @rep set<E> future;  // future iteration elements
}
```

Figure 5. Set, bag and iterator specification samples.

We use the notation `:x` inside a path expression to denote a set of intermediate objects during an access path traversal. We write `+` and `-` for set union and difference respectively.

Example 2 (Legend)**.** *The following paths are derived from the abstract store in Figure 6:*

- *`a.rev(src)` represents the outgoing edges of `a`: $\{2, 3\}$.*
- *`a.rev(dst)` represents the incoming edges of `a`: $\emptyset$.*
- *`a.rev(src).dst + a.rev(dst).src` represents all of the graph nodes adjacent to `a`: $\{b, c\}$.*
- *`a.rev(src):x.dst.lt` sets the temporary variable `x` to the edge from `a` to `lt`: $x = \{3\}$ (2 is not on a path from `a` to `lt`).*

Specifying Abstract Locks. A `@locks` annotation defines the set of abstract locks a method should acquire by a set of path expressions. To keep specifications succinct, access paths expressions in `@locks` stand for all of their prefixes (e.g., `n.rev(src).dst` stands for `n`, `n.rev(src)`, and `n.rev(src).dst`).

We call a node referenced by `n` along with its incident edges and adjacent nodes the immediate neighborhood of `n`. We specify the set of locks for such a neighborhood by `@locks(n.rev(src).dst, n.rev(dst).src)`.

Example 3 (Commutativity via abstract locks)**.** *The `removeNode` method specifies locks for the immediate neighborhood of the node being removed. A call to `removeNode(c,LOCK)` attempts to lock the `Node` object `c`, the `Edge` objects referencing `c` via the `src` field or the `dst` field (the edges incident to `c`), and the `Node` objects that are neighbors of `c`. This ensures the concurrent method calls `removeNode(c,LOCK)` and `removeNode(e,LOCK)` will not cause their respective iterations to abort, since the immediate neighborhoods of `c` and `e` do not overlap.*

Specifying Method Semantics. An `@op` annotation defines the semantics of a method by a simple imperative language. The language allows a sequence of statements, using set expressions over method parameters, static fields, and temporary variables. Expressions of the form `a in b`, `a notIn b`, and `isEmpty(a)`, test whether `a` is contained in `b`, `a` is not contained in `b`, and whether `a` is an empty

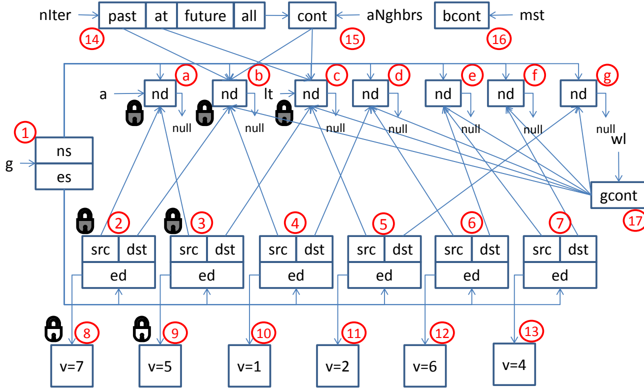

Figure 6. An abstract store arising at L2, using as input the graph from Figure 1. Object are shown by rectangles sub-divided by their fields; circles are used to name objects; locks show objects contained in the global `GaloisRuntime.locks` set. We label `Node` objects by the same labels used in Figure 1 and other objects by a running index.

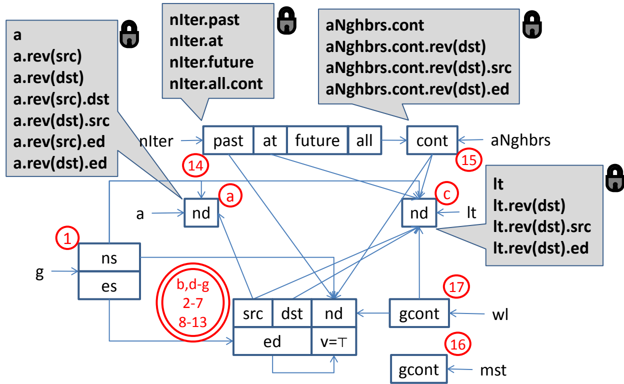

Figure 7. A shape graph obtained by applying hierarchy summarization abstraction to the store in Figure 6. Grey boxes represent sets of locked objects. `v=`$\top$ indicates that the numeric value of `v` has been abstracted away.

set, respectively. (We overload these expressions to treat reference variables as singleton sets.) Statements of the form `a += exp` and `a -= exp` are shorthand for `a = a + exp` and `a = a - exp`, respectively. `choose(exp)` non-deterministically chooses an object from a set denoted by `exp`.

2.4 Optimization Opportunities

Our static analysis enables the following optimizations.

Eliminating Usage of Concurrent Data Structures. The following conditions allow replacing a concurrent implementation of a data structure by a sequential implementation: the data structure is iteration private, or the data structure is never modified. We use a purity analysis [33] to discover objects that are never modified inside an iteration (such as `Weight` in the running example).

Reducing Rollback Logging. Logging inverse method calls for iterations that commit represents wasted work, as the log is cleared when the iteration commits and the logged method calls are never used.

Our static analysis finds a minimal set of *failsafe points* — program locations in the client program such that an iteration reaching them cannot abort. The analysis computes an under-approximation of the set of objects that are always locked at a program location. If the set of locks computed for a location `L` subsumes the set of locks computed for all locations reachable from `L`, then `L` is a failsafe point. An iteration reaching a failsafe point will never fail to acquire a lock and therefore cannot abort. We eliminate logging inverse actions for method calls appearing after a failsafe point.

If no method call before a failsafe point modifies shared data structures, rollback logging is not needed anywhere in the iteration. Algorithms with this property are called *cautious* algorithms [26].

Eliminating Redundant Locking. Our analysis can also be used to find method calls for which all locks have already been acquired by preceding method calls. Lock acquisitions can be eliminated for these calls. Furthermore, our analysis finds user-defined objects "dominated" by other locked objects, *i.e.*, objects that can only be accessed after a unique abstract lock is acquired. We eliminate lock operations for such objects as well.

2.5 Optimizing the Running Example by Static Analysis

We develop a sound static analysis to automatically infer available optimizations of the kind discussed above. The input to our analysis is a Java program with a single parallel loop, given by the `foreach` construct, operating over a library of specified boosted data structures. The output of our analysis is an assignment of option flags to each ADT method call and a list of (user-defined) types that do not need "transactional" protection.

The core component of our analysis is a shape analysis that under-approximates the set of objects that are always locked at a program point. Intuitively, our analysis abstracts stores into bounded-size shape graphs by collapsing all objects not referenced by variables together and recording for each root object a set of path expressions denoting the set of locked objects.

Figure 7 shows a shape graph obtained by applying our abstraction to the store in Figure 6. The object labeled by a double-circle shows the set of collapsed objects. The grey box pointing to **a** expresses the fact that the immediate neighborhood of **a** is locked, along with the `Weight` objects referenced by its incident edges. This shape graph represents an intermediate invariant inferred by our analysis at `L2`. The full invariant is given by a set of the shape graphs at that point, at the fixpoint.

Below, we provide sample invariants that our analysis infers for Figure 2 and the corresponding path expressions denoting sets of objects all of which are locked:

*Inv*1: At `L2`, the immediate neighborhood of `a` is locked:
`a + a.rev(dst).src + a.rev(src).dst`.

*Inv*2: At `L4`-`L9`, the immediate neighborhoods of `a` and `lt` are locked:
`a + a.rev(dst).src + a.rev(src).dst +`
`lt + lt.rev(dst).src + lt.rev(src).dst`.

*Inv*3: At `L2` and `L6`, all graph nodes accessible by the iterator `nIter` (past, present, and future iterations) are locked: `nIter.past + nIter.at + nIter.future`.

*Inv*4: At `33`, `47`, `50`, and `53`, the edges referenced by `e` and `an` and the nodes they reference are locked:
`e + an + e.src + e.dst + an.src + an.dst`.

*Inv*1 is part of the invariant needed to prove that `L4` is a failsafe point (before executing the statement). *Inv*1 needs to be maintained from `L2` and on. It is also used to eliminate locking in lines `32`-`33`. *Inv*2 helps establish `L4` as a failsafe point, since all accesses to nodes and edges in the second loop are to objects known to be locked. Also, it helps eliminate redundant locking at `L9`. *Inv*3 helps

establish the failsafe point at `L4` by the fact that the node referenced by `n` is locked at 46 and 48, and eliminate locking at 32. Finally, *Inv*4 establishes that the calls to `getEdgeData` and `setEdgeData` in lines 33, 47, 50, and 53 do not lock new objects.

Additionally, our analysis infers that `Weight` objects are read-only, which enables eliminating all lock operations and backup copy maintenance for them. Points `L7`, `L8`, `L10`, `L11` are after the failsafe point and do not require storing inverse actions. At `L10`, the calls to the `add` method of `Bag` do not require acquiring locks, and trivially commute.

We apply these optimizations to the code of Figure 2 by setting the `LOCK` option at `L1` and `L3`, which eliminates rollback logging, and setting the `NONE` option in all other calls, eliminating both abstract locking and rollback logging.

This implementation of Boruvka's algorithm is cautious: our analysis infers that the failsafe point is `L4` and that no modifications are made to the graph between `L1` and `L4`. If we remove the statement at `L3`, the failsafe point is at `L5`, which requires logging an inverse method call for `g.removeEdge(a, lt)`.[4]

3. A Shape Analysis for Graph Programs

This section presents our static analysis for enabling the optimizations described in previous sections. Our analysis considers only sequential executions, but the inferred properties apply to concurrent executions as well. We use a result by Filipovic et al. [8] and the fact that our concurrent executions are strictly serializable to formally justify this [30].

The core component of the analysis is a shape analysis that under-approximates the set of objects that are always locked, at each program location. This section is organized as follows: (1) we discuss the class of programs and stores that our shape analysis addresses; (2) we define Canonical Abstraction [32] and partial join [23] in our setting; (3) we define ***H**ierarchy **S**ummarization **A**bstraction* (*HSA*); (4) we present a technique for discovering predicates relevant to our analysis; (5) we explain how the results of the shape analysis are used; (6) we contrast our abstraction with ***B**ackward **R**eachability **A**bstraction* (*BRA*), a commonly used form of shape abstraction; (7) we discuss how our analysis can aid the programmer by providing non-cautiousness counterexamples; and finally (8) we discuss limitations of our analysis.

3.1 A Class of Programs and Stores

We analyze Java programs (excluding recursive procedures) where the implementation of specified data structures is replaced by the abstract fields in the `@rep` annotations and the semantics of methods is given by the `@op` annotations.

Figure 8 defines stores in terms of pointer fields and set-valued fields defined by the `@rep` annotations. We define the meaning of path expressions (recursively), which denote sets of objects reachable from a variable by following fields in specified directions and going through specified variables. The last definition in Figure 8 provides the meaning of variables assigned to intermediate objects along path expressions, such as `f.rev(src):eft.dst.t`.

Bounded-depth Hierarchical Stores. We define the set of types reachable from an object o (by forward paths) to be the set of types of all objects in $[\![o.p]\!]$ for all path expressions p.

This paper focuses on the class of *bounded-depth hierarchical stores* — stores where the set of types reachable from $[\![\texttt{o.f}]\!]$ is a proper subset of the set of types reachable from o, for every object o and field f. Such stores are acyclic — the length of any unidirectional path, i.e., a path where all fields are either forward or reverse, is linearly bounded by the number of program types.

[4] Swapping `L4` and `L5` makes the code cautious once again, but breaks sequential correctness, since in the Galois library it is illegal to remove an edge while iterating over the neighbors of a node incident to it.

3.2 Canonical Abstraction and Partial Join

We implement our shape analysis using the TVLA system [22], which allows defining stores by first-order predicates, program statements by first-order transition formulae (formulae relating the values of predicates after a statement to those before), and abstract states by first-order *abstraction predicates*. The system automatically generates sound abstract operations and transformers, yielding a sound abstract interpretation for a given program.

TVLA uses Canonical Abstraction [32], which abstracts stores into 3-valued logical structures. To focus our presentation on the important details of our analysis, we simplify our description of TVLA's abstraction and use *shape graphs* for abstract states instead of 3-valued structures.

Definition 3.1 (Shape Graph)**.** *Let $P = AP \cup NAP$ be a set of predicates consisting of two disjoint sets of unary predicates called* abstraction predicates *(AP) and* non-abstraction predicates *(NAP). A shape graph G is a tuple (N^G, P^G, E^G) where N^G is a set of abstract objects, $P^G : N^G \to 2^P$ assigns predicates to objects, and $E^G : OFld \cup SFld \to N^G \times N^G$ is a set of* may-edges *for each field. We denote the set of shape graphs over P by ShapeGraph[P].*

We call the set of abstraction predicates assigned to an abstract node $v \in N^G$ its *canonical name*: $CName(v) \overset{\text{def}}{=} P^G(v) \cap AP$. A shape graph G is *bounded* if no two abstract nodes have the same canonical name. This means that the number of abstract nodes in a bounded shape graph is *exponentially bounded* by the number of abstraction predicates.

We define the abstraction function $\beta[P] : Store \to BGraph[P]$, which maps a store $\sigma = (S^\sigma, H^\sigma)$ into a bounded shape graph G as follows. We use the helper function $P^\sigma : T_O \to 2^P$, which evaluates the predicates in P for each object, and $\mu^{\sigma,G} : T_O \to N^G$, which maps store objects having an equal canonical name to an abstract node representing their equivalence class in G. The predicate assignment function assigns to abstract nodes the predicates common to all objects they represent, and a field edge exists between two abstract nodes if there exist two objects represented by the abstract nodes that are related by that field.

$$\mu^{\sigma,G}(o_1) = \mu^{\sigma,G}(o_2) \iff P^\sigma(o_1) \cap AP = P^\sigma(o_2) \cap AP$$

$$\begin{aligned} P^G(n) &= \bigcap_{o \text{ s.t. } n=\mu^{\sigma,G}(o)} P^\sigma(o) \\ E^G(f) &= \{(n_1,n_2) \mid \exists o_1, o_2 \colon n_1 = \mu^{\sigma,G}(o_1), n_2 = \mu^{\sigma,G}(o_2). \\ &\quad \begin{cases} o_2 = H^\sigma(f)(o_1), & f \in OFld; \\ o_2 \in H^\sigma(f)(o_1), & f \in SFld. \end{cases} \\ &\quad \} \,. \end{aligned}$$

We say that a shape graph G' *subsumes* a shape graph G, written $G \sqsubseteq G'$, if there exists an onto function $\mu^{G,G'} : N^G \to N^{G'}$, such that $P^G(n) \supseteq P^{G'}(\mu^{G,G'}(n))$ for all $n \in N^G$, and $(n_1, n_2) \in E^G(f)$ implies that $(\mu^{G,G'}(n_1), \mu^{G,G'}(n_2)) \in E^{G'}(f)$ for all $n_1, n_2 \in N^G, f \in OFld \cup SFld$.

The meaning of a shape graph G is given by the function $\gamma[P] : ShapeGraph[P] \to 2^{Store}$ defined as $\gamma[P](G) = \{\sigma \mid \beta[P](\sigma) \sqsubseteq G\}$.

We say that two shape graphs G and G' are *congruent* if there exists a bijection between their sets of abstract nodes $\mu^{G,G'} : N^G \to N^{G'}$, which preserves the abstraction predicates: $P^G(n) \cap AP = P^{G'}(\mu^{G,G'}(n)) \cap AP$ for all $n \in N^G$. Two congruent shape graphs G and G' can be subsumed by a congruent shape graph $G'' = G \sqcup G'$, by intersecting corresponding predicate values and taking the union of corresponding edges using the bijections

Stores

T_O	Objects
$\mathit{Stack} : \mathit{PVar} \to T_O \cup$ $\quad \mathit{SVar} \to 2^{T_O} \cup$ $\quad \mathit{BVar} \to \{\texttt{T}, \texttt{F}\}$	Stacks
$\mathit{Heap} : (T_O \times \mathit{OFld}) \to T_O \cup$ $\quad (T_O \times \mathit{SFld}) \to 2^{T_O}$	Heaps
$\mathit{Store} : \mathit{Stack} \times \mathit{Heap}$	Stores
$\sigma = (S^\sigma, H^\sigma)$	Store notation

Semantics of Path Expressions

$[\![\mathit{Path}]\!] : \mathit{Store} \to 2^{T_O}$

Base case. Variables:

$$[\![x]\!](\sigma) = \begin{cases} \{S^\sigma(x)\}, & x \in \mathit{PVar}; \\ S^\sigma(x), & x \in \mathit{SVar}. \end{cases}$$

Inductive case. $p \in \mathit{Path}$, $[\![p]\!](\sigma)$ is known, and e is a field or variable:

$$[\![p.e]\!](\sigma) = \begin{cases} [\![p]\!](\sigma) \cap [\![e]\!](\sigma), & e \in \mathit{PVar} \cup \mathit{SVar}; \\ \{H^\sigma(o, e) \mid o \in [\![p]\!](\sigma)\}, & e \in \mathit{OFld}; \\ \bigcup_{o \in [\![p]\!](\sigma)} H^\sigma(o, e), & e \in \mathit{SFld}; \\ \{o \in T_O \mid H^\sigma(o, f) \in [\![p]\!](\sigma)\}, & e = \texttt{rev}(f), f \in \mathit{OFld}; \\ \{o \in T_O \mid H^\sigma(o, f) \cap [\![p]\!](\sigma) \neq \emptyset\}, & e = \texttt{rev}(f), f \in \mathit{SFld}. \end{cases}$$

For an object $o \in T_O$ and path $p \in \mathit{Path}$, we define
$[\![o.p]\!](\sigma) = \textbf{let } y$ be fresh, $\sigma' = (S^\sigma | y \mapsto o, H^\sigma)$ **in** $[\![y.p]\!](\sigma')$

Meaning of intermediate variables:
The expression $x.p{:}v.q$ assigns to v the set of objects that are both on a path from x to q and in $x.p$. For $x \in \mathit{Var}, v \in \mathit{SVar}, p, q \in \mathit{Path}$
$[\![v]\!](\sigma) = [\![x.p]\!](\sigma) \cap \{o \in T_O \mid [\![o.q]\!](\sigma) \neq \emptyset\}$.

Figure 8. Stores and semantics of path expressions.

$\mu^{G'',G} : N^{G''} \to N^G$ and $\mu^{G'',G'} : N^{G''} \to N^{G'}$:

$$\begin{array}{rcl} N^{G''} & = & N^G \\ P^{G''}(n) & = & P^G(\mu^{G'',G}(n)) \cap P^{G'}(\mu^{G'',G'}(n)) \\ (n_1, n_2) \in E^{G''}(f) & \Leftrightarrow & (\mu^{G'',G}(n_1), \mu^{G'',G}(n_2)) \in E^G(f) \text{ or} \\ & & (\mu^{G'',G'}(n_1), \mu^{G'',G'}(n_2)) \in E^{G'}(f) \ . \end{array}$$

We use TVLA's partial join operator [23], which merges congruent shape graphs into a single shape graph, and keeps non-congruent shape graphs in a set:

$$\{G\} \sqcup \{G'\} \stackrel{\text{def}}{=} \begin{cases} \{G \sqcup G'\}, & G \text{ and } G' \text{ are congruent;} \\ \{G, G'\}, & \text{else.} \end{cases}$$

The abstraction of a set of stores $\alpha[P] : 2^{\mathit{Store}} \to 2^{\mathit{BGraph}}[P]$ is defined as $\alpha[P](\Sigma) = \bigsqcup_{\sigma \in \Sigma} \beta[P](\sigma)$.

3.3 Hierarchy Summarization Abstraction

Our abstraction is defined relative to a set of *abstraction paths*, denoted by *AbsPaths*, which represent possible paths from variables to locked objects. The next subsection discusses a technique to discover a set of useful abstraction paths for a set of data structures.

Let $\sigma = (S^\sigma, H^\sigma)$ be a store. For a pointer variable x and an abstraction path p, we define a unary predicate expressing the fact that v is a root object referenced by x and **all** objects reachable from it by the path p are locked:

$$\mathit{ForwardReach}[x, p](v) \stackrel{\text{def}}{=} [\![x]\!]^\sigma = \{v\} \wedge [\![x.p]\!]^\sigma \subseteq [\![\texttt{locks}]\!]^\sigma \ .$$

We encode hierarchy summarization abstraction via shape graphs and the set of predicates P^{HSA}, shown in Table 1, and the abstraction paths in Table 2. Since a pointer variable points to at most one node, the number of abstract nodes in a bounded shape graph $G \in \mathit{BGraph}[P^{HSA}]$ is equal to at most the number of heap roots + 1 (in the case where there exist non-root objects). The canonical names in such a shape graph are the sets of aliased pointer variables. We call such sets *aliasing configurations*. In practice, the average number of different aliasing configurations discovered by our analysis is a small constant (≈6), which means that the set of bounded shape graphs our analysis explores is linear in the number of program locations.

Figure 7 shows the result of applying $\beta[P^{HSA}]$ to the store in Figure 6 and the predicates in Table 1. Heap roots are labeled with path expressions that denote the sets of objects that are reachable from them and are definitely locked. At L2, we would expect that node **a**, its neighbors, and the edges connecting **a** are locked. This is specified by the path expressions labeling node **a**. For example, `a.rev(src).dst`, `a.rev(dst).src` refer to all the neighbors of **a**. Additionally, the current element that we are iterating over is node **c**, which is the lightest neighbor of **a**; this node has its single incoming edge and edge data locked. All other (non-root) nodes, edges, and `Weight` objects are collapsed by our abstraction.

Predicates	Meaning
Abstraction Predicates	
$\{x(v) \mid x \in \mathit{Var}\}$	x references v
Non-abstraction Predicates	
$\{\mathit{ForwardReach}[x, p](v) \mid x \in \mathit{Var}, p \in \mathit{AbsPaths}\}$	Hierarchy summarization predicates

Table 1. P^{HSA} predicates for hierarchy summarization abstraction.

Type	Abstraction Paths
`Graph`	$\texttt{es}$, $\texttt{es.src}$, $\texttt{es.dst}$, $\texttt{ns}$, $\texttt{ns.}\overleftarrow{\texttt{src}}$, $\texttt{ns.}\overleftarrow{\texttt{dst}}$, $\texttt{ns.}\overleftarrow{\texttt{dst}}\texttt{.ed}$, $\texttt{ns.}\overleftarrow{\texttt{dst}}\texttt{.ed}$, $\texttt{ns.}\overleftarrow{\texttt{src}}\texttt{.dst}$, $\texttt{ns.}\overleftarrow{\texttt{dst}}\texttt{.src}$, $\texttt{ns.nd}$, $\texttt{ns.}\overleftarrow{\texttt{src}}\texttt{.dst.nd}$, $\texttt{ns.}\overleftarrow{\texttt{dst}}\texttt{.src.nd}$, $\texttt{es.ed}$, $\texttt{es.src.nd}$, $\texttt{es.dst.nd}$
`Node`	$\texttt{a}$, $\texttt{lt}$, $\texttt{n}$, $\overleftarrow{\texttt{src}}$, $\overleftarrow{\texttt{dst}}$, $\overleftarrow{\texttt{src}}\texttt{.dst}$, $\overleftarrow{\texttt{dst}}\texttt{.src}$, $\texttt{nd}$ $\overleftarrow{\texttt{src}}\texttt{.ed}$, $\overleftarrow{\texttt{dst}}\texttt{.ed}$, $\overleftarrow{\texttt{src}}\texttt{.dst.nd}$, $\overleftarrow{\texttt{dst}}\texttt{.src.nd}$
`Edge`	`e`, `an`, `src`, `dst`, `ed`, `src.nd`, `dst.nd`
`Weight`	$\overleftarrow{\texttt{ed}}$, $\overleftarrow{\texttt{ed}}\texttt{.src}$, $\overleftarrow{\texttt{ed}}\texttt{.dst}$, $\overleftarrow{\texttt{ed}}\texttt{.src.nd}$, $\overleftarrow{\texttt{ed}}\texttt{.dst.nd}$
`Set`	$\texttt{cont}$, $\texttt{cont.}\overleftarrow{\texttt{src}}$, $\texttt{cont.}\overleftarrow{\texttt{dst}}$, $\texttt{cont.}\overleftarrow{\texttt{src}}\texttt{.dst}$, $\texttt{cont.}\overleftarrow{\texttt{dst}}\texttt{.src}$, $\texttt{cont.}\overleftarrow{\texttt{src}}\texttt{.ed}$, $\texttt{cont.}\overleftarrow{\texttt{dst}}\texttt{.ed}$, $\texttt{cont.}\overleftarrow{\texttt{src}}\texttt{.dst.nd}$, $\texttt{cont.}\overleftarrow{\texttt{dst}}\texttt{.src.nd}$, $\texttt{cont.nd}$
`GSet`	`gcont`, `gcont.nd`
`GBag`	`bcont`
`Iterator`	`all`, `all.cont`, `all.cont.nd`, `past`, `at`, `future`, `past.nd`, `at.nd`, `future.nd`

Table 2. Abstraction paths for the running example. We omit Java Generics parameters when no confusion is likely.

3.4 Predicate Discovery

We now describe heuristics for generating the set of abstraction paths from the data structures in a program, and show how it finds paths expressing the invariants described in Section 2. Our technique constructs paths in three phases: (a) building the *type dependence graph*, (b) discovering *variable-to-lock* paths in method specifications, and (c) combining variable-to-lock paths and all forward paths in the type dependence graph.

Definition 3.2 (Type Dependence Graph)**.** *A* type dependence graph *for a program, contains a* type node N_T *for each program type* T*, labeled by the set of program variables of that type; and a* field edge *from type node* N_T *to type node* $N_{T'}$*, labeled by a field of type* T' *or* $set\langle T'\rangle$ *declared in type* T*.*

Figure 9 shows the type dependence graph for Figure 2. For the rest of this section, we fix the set of variables and fields, and define the set of well-formed path expressions, *WFPath*.

Definition 3.3 (Well-formed Path Expressions)**.** *Define the* type-node pair *of a path expression element as follows:* $TNPair(x) = (N_T, N_T)$ *for a variable* x *of type* T*;* $TNPair(f) = (N_T, N_{T'})$ *for a field* f *of type* T' *or* $set\langle T'\rangle$ *declared in a type* T*; and* $TNPair(\overleftarrow{f}) = (N_{T'}, N_T)$ *for a reversed field expression* $\overleftarrow{f}$*, if* $TNPair(f) = (N_T, N_{T'})$*.*

Let p *be a path expression* $e_1.e_2.\ldots.e_k$ *and let the corresponding sequence of type-node pairs be* $(N_1, N_1'), \ldots, (N_k, N_k')$*. We say that* p *is* well-formed *if the sequence of type-nodes* $N_1, N_1', \ldots, N_k, N_k'$ *is an undirected path in the type dependence graph. We define the type-node pair of* p *to be* $TNPair(p) = (N_1, N_k')$*.*

Example 4. *For example,* $\texttt{nIter.at.}\overleftarrow{\texttt{gcont}}\texttt{.wl}$ *is well-formed, whereas* `g.nd` *and* $\overleftarrow{\texttt{ed}}\texttt{.es}$ *are not.*

In the sequel, we consider only well-formed path expressions. We say that a path expression p contains a cycle if the corresponding path in the type dependence graph contains a cycle. A forward path is a (well formed) path expression that contains no reversed field sub-expressions.

Definition 3.4 (Forward Closure)**.** *The* forward closure *of a path expression* p*, written* $Forward(p)$*, is the set of all path expressions of the form* $p.p'$ *where* p' *is a forward path not containing program variables (*$p.p'$ *is well-formed) and* p' *does not introduce cycles other than ones already contained in* p*. The forward closure of a type* T *is the set of all forward paths starting from type* T*, not containing program variables.*

Path closures of sets of path expressions and types are obtained by taking the union of the closures of all set members.

Example 5. $Forward(\texttt{Edge}) = \{\texttt{ed}, \texttt{src}, \texttt{dst}, \texttt{src.nd}, \texttt{dst.nd}\}$ *and* $Forward(\texttt{an.src.}\overleftarrow{\texttt{dst}}) = \{\texttt{an.src.}\overleftarrow{\texttt{dst}}, \texttt{an.src.}\overleftarrow{\texttt{dst}}\texttt{.ed}\}$*.*

The forward closure of the types in the type dependence graph represent data access patterns where a sequence of method calls is used to obtain an object of type T from an object of a type T' higher in the hierarchy. For example, in lines 32–33 of Figure 2, a sequence of method calls is used to obtain an edge from the graph and a `Weight` from an edge. In particular, the forward closure gives us the paths needed to express *Inv*3 and *Inv*4.

However, these paths ignore the effect of methods, which create more intricate paths, such as the ones needed for *Inv*1 and *Inv*2. Those are discovered by "summarizing" method specifications, as explained next.

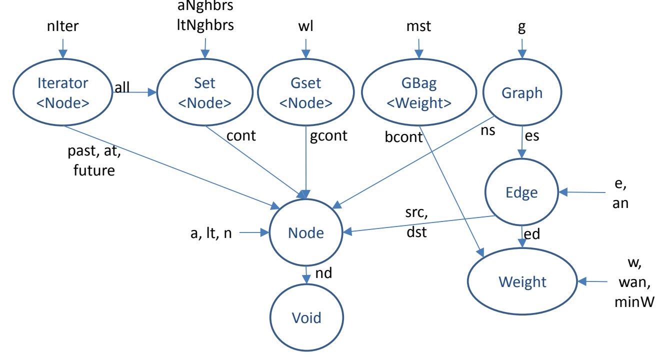

Figure 9. Type dependence graph for Figure 2.

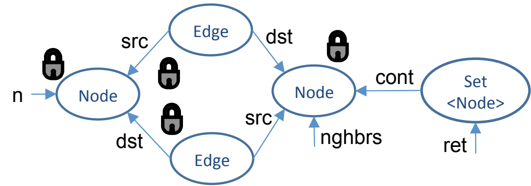

Figure 10. Footprint graph for `Graph.getNeighbors`. A lock is shown next to each node labeled by `locks`.

3.4.1 Discovering Paths in Method Footprints

We now explain how to find variable-to-lock paths, which represent possible paths between objects referenced by the method parameters (and returned value) and objects accessed by the `@locks` specification, after the `@op` specification "executes".

To find these paths, we construct a *footprint graph* for each method. Intuitively, this graph represents the set of objects accessed by the method, sometimes referred to as the "footprint" of the method. The idea of "footprint analysis" was defined by Calcagno et al. [3] to infer method preconditions and postconditions. We put this idea to use for a different purpose.

We create a footprint graph by the following steps:

Handling statements We interpret the statements in `@op` in the order they appear. For each statement, we create a graph representing every path expression on the right-hand side of an assignment. This is done by creating a new node for each position in the expression, connecting them by the respective fields, and labeling nodes by the variables along the expression. If the left-hand side of the assignment is a pointer or set variable (`locks`), we use it to label the last node of each path graph. If it is a field of the type containing the method, we create a node of that type labeled by `this` and connect an edge field from that node to the last node of every path graph created for the right-hand side expression.

Creating `@locks` paths We create path graphs for all path expressions in `@locks` that do not already appear in `@op`.

Merging We merge nodes labeled by a common (pointer or set) variable.

Setting `locks` We label every node matching a path expression in `@locks` by `locks`.

Example 3.5. *Figure 10 shows the footprint graph for the* `getNeighbors` *method of* `Graph`*. The top node represents the outgoing edges of* `n`*, the lower node represents the incoming edges*

Type	Variable-to-Lock Paths
Graph	es, es.src, es.dst, ns, $\texttt{ns.}\overleftarrow{\texttt{src}}$, $\texttt{ns.}\overleftarrow{\texttt{dst}}$, $\texttt{ns.}\overleftarrow{\texttt{src}}\texttt{.dst}$, $\texttt{ns.}\overleftarrow{\texttt{dst}}\texttt{.src}$
Node	Var(Node), $\overleftarrow{\texttt{src}}$, $\overleftarrow{\texttt{dst}}$, $\overleftarrow{\texttt{src}}\texttt{.dst}$, $\overleftarrow{\texttt{dst}}\texttt{.src}$
Edge	Var(Edge), src, dst
Weight	$\overleftarrow{\texttt{ed}}$, $\overleftarrow{\texttt{ed}}\texttt{.src}$, $\overleftarrow{\texttt{ed}}\texttt{.dst}$
Set<Node>	cont, $\texttt{cont.}\overleftarrow{\texttt{src}}$, $\texttt{cont.}\overleftarrow{\texttt{dst}}$, $\texttt{cont.}\overleftarrow{\texttt{src}}\texttt{.dst}$, $\texttt{cont.}\overleftarrow{\texttt{dst}}\texttt{.src}$
GSet	gcont
GBag<E>	$\emptyset$

Table 3. Variable-to-Lock paths for the running example. Var(T) denotes an arbitrary variable to an object of type T.

of n*. Both are connected to some neighbor of* n*. The node on the right represents the returned set containing the neighbors of* n*. We use this graph to obtain paths expressing that* getNeighbors *has the effect of locking the immediate neighborhood of* n*.*

We define the function $\mathit{VarToLock} : \mathit{TName} \to 2^{\mathit{WFPath}}$ associating a set of variable-to-lock paths with each program type.

We create a set of variable-to-lock paths for every type node from all footprint graphs as follows. For each footprint graph, we take all the acyclic non-empty paths from a node labeled by a method parameter (including this and the return parameter ret) to any node labeled by locks. We associate these paths with the type node corresponding to the type of the parameter. We denote the set of variable-to-lock paths of type T by $\mathit{VarToLock}(T)$.

Table 3 shows the variable-to-lock paths that we get for the running example. These paths enable us to express *Inv*1 and *Inv*2.

We combine the sets of paths defined earlier to obtain the set of abstraction paths:

$$\mathit{AbsPaths} \stackrel{\text{def}}{=} \bigcup_{t \in \mathit{TName}} \mathit{Forward}(t) \cup \mathit{Forward}(\mathit{VarToLock}(t)) \quad .$$

Here, expressions of the form Var(T) appearing in $\mathit{VarToLock}(t)$ are substituted by the set of paths $\{x \in \mathit{Var} \mid x \text{ is of type } T\}$.

3.5 Putting it All Together

Our overall static analysis consists of the following stages:

Preprocessing We use a lightweight purity analysis [33] to detect objects that do not require concurrency control and fields that are never used inside the parallel loop, e.g., the initSrc and initDst fields of Weight. The remainder of the analysis does not consider path expressions in @locks containing unused fields and sets the opt flags of read-only objects to NONE.

Shape Analysis We execute a forward shape analysis using hierarchy summarization abstraction and TVLA-generated abstract transformers. The fixpoint is a set of bounded shape graphs at every program location.

Finding Redundant Locks We use abstract operations in TVLA to conservatively check whether every shape graph at a program location represents stores that lock all objects defined by a @locks specification of a method executing at that location. If so, we set the opt argument of that method call to UNDO (if it was not already set to NONE).

Finding Failsafe Points We perform a backward BFS traversal over the CFG (control flow graph) to find earliest program locations where all following method calls are labeled by NONE or UNDO (meaning they do not acquire locks). These program locations are the program failsafe points, We set the optimization argument of all method calls dominated by failsafe points to NONE.

Predicates	Meaning
Abstraction Predicates	
$\{x(v) \mid x \in \mathit{Var}\}$	x references v
$\{\mathit{BackwardReach}[x,p](v) \mid x \in \mathit{Var}, p \in \mathit{AbsPaths}\}$	Backward-reachability predicates

Table 4. Predicates for backward-reachability abstraction.

3.6 Backward Reachability Abstraction

A common abstraction idiom for shape abstraction uses *coloring*, which records a set of unary (object-)predicates with **every object** in the store. These predicates are used to partition the set of objects into equivalence classes. Examples are Canonical Abstraction [32], Boolean heaps [28], Indexed predicate abstraction [20], and generalized typestates [21].

These abstractions typically employ *backward reachability predicates* that use paths in the heap to relate objects to variables. For example, most TVLA-based analyses and analyses using Boolean heaps distinguish between disjoint data structure regions (e.g., list segments and sub-trees) by using transitive reachability from pointer variables. Indexed predicate abstraction [20] uses predicates that assert that cache clients are contained in one of two lists (sharer_list and invalidate_list). Lam et al. [21] use set containment predicates as the generalized typestate of an object.

We call such abstractions *bottom-up*, since they record properties of objects deep in the heap with respect to (shallow) root objects. These abstractions achieve high precision as they express every Boolean combination of intersection and union of objects satisfying the unary predicates. However, the size of an abstracted store can be exponential in the number of predicates, which might lead to state space explosion in cases where objects satisfy many different subsets of predicates.

We define backward reachability abstraction by using the set of abstraction paths presented earlier to define backward reachability predicates. For a pointer variable x and an abstraction path p, we define a unary predicate expressing the fact that v is a locked object reachable from x by the path p:

$$\mathit{BackwardReach}[x,p](v) \stackrel{\text{def}}{=} v \in [\![\texttt{locks}]\!]^{\sigma} \cap [\![x.p]\!]^{\sigma} \quad .$$

We obtain a backward reachability abstraction $\beta[P^{\mathit{BRA}}]$ from the predicates shown in Table 4. *BRA* is strictly more precise than *HSA*. However, it can be very expensive — the number of abstract nodes in a shape graph obtained by $\beta[P^{\mathit{BRA}}]$ can be exponential in the number of backward-reachability predicates. State space explosion manifests when stores create overlaps between different interacting sets (set fields), which is often the case in our programs. Applying $\beta[P^{\mathit{BRA}}]$ to the store in Figure 6, will conflate all objects not locked and not referenced by a program variable. Compared to Figure 7, Edge objects 2 and 3, for example, will remain needlessly distinguished. Situations such as iterating over the neighbors of a node, exploring multiple neighborhoods simultaneously or sharing objects between multiple collections, cause the number of useless distinctions to increase.

3.7 Producing Non-Cautiousness Counterexamples

When the code of a parallel loop body is not cautious, our analysis can sometimes provide a counterexample to demonstrate the violation of the cautious property at appropriate program points. To find such counterexamples, we assume the small scope conjecture [14], which says that counterexamples usually manifest in small graphs.

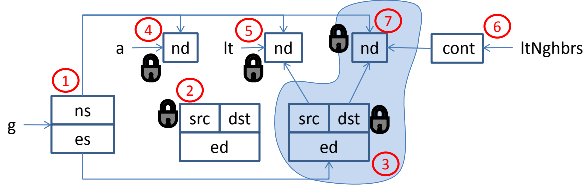

Figure 11. A counterexample at location `L5` for the non-cautious implementation of BVK.

A graph with three nodes and two edges is sufficient to provide us with a counterexample for the case of BVK, as shown in Figure 11. The region of the graph where the violation happens is highlighted. This is the smallest counterexample found by our analysis, taking about 300 seconds to produce.

3.8 Limitations

We recognize the following limitations of our analysis.

Bounded-depth hierarchy. As discussed at the beginning of this section, we assume a class of stores where a finite-depth hierarchy property exists. This allows us to ensure a bound on the number of hierarchy summarization paths used to define our abstraction. This precludes us from handling benchmarks where data structures such as lists and trees are explicitly manipulated (and cannot be abstracted away by a `@rep` specification). Generalizing our analysis to handle recursive data structures may be done by considering abstraction paths with regular expressions over the pointer fields of the data structure.

Temporary violation of invariants. Our abstraction is geared to infer invariants of the form $\forall o.R(o) \implies p(o)$ where $R(o)$ expresses a heap region (by abstraction paths) and $p(o)$ is a property we wish to summarize for the objects in the region $R(o)$ (the is-locked property in our analysis). When the property p is temporarily violated for the objects in $R(o)$ and then restored, our analysis is not able to restore the invariant. For example, assume an invariant $\forall o.R(o) \implies p(o)$ holds at program point 1. Then a point 2 a single object in $R(o)$, referenced by a pointer variable `x`, is made to have $\neg p(o)$ and at point 3 it is removed from $R(o)$. In order to regain the invariant $\forall o.R(o) \implies p(o)$ at point 3, we may need to refine our abstraction in order to express an invariant such as $\forall o.(R(o) \wedge \neg \mathtt{x}(o)) \implies p(o)$.

4. Experimental Evaluation

The shape analysis described in Section 3 was implemented in TVLA, and used to optimize four benchmarks from the Lonestar suite [15]. These benchmarks were chosen because they exhibit very diverse behavior. We describe them below.

- BVK: Boruvka's MST algorithm. This benchmark adds and removes nodes and edges from a graph.
- DMR: Delaunay mesh refinement. This benchmark uses iterative refinement to produce a quality mesh. In each iteration, a neighborhood of a bad triangle, called the *cavity* of that triangle, is removed from the mesh and replaced with new triangles. DMR uses a large number of collections with intricate patterns of data sharing, so it is a "stress test" for the analysis.
- SP: Survey propagation, a heuristic SAT solver. Most iterations only update node labels, but once in a while, an iteration removes a node (corresponding to a "frozen variable" [2]) and its incident edges.

Prog.	IR Size	Graph Calls	Set Calls	Field Acc.	Optimal
BVK	340	17/20	4/4	23/23	✓
DMR	1,168	26/30	30/30	164/164	✓
SP	925	32/34	16/16	123/123	✓
PFP	479	6/8	3/3	28/28	✓

Table 5. Program characteristics and static analysis results. x/y measures Optimized/Total.

Analysis	Total SGs	Avg. # Abs. Nodes	Avg. # SGs CFG Location	Time (sec)
BVK				
HSA	13,594	9	6.25	6
BRA	412,862	15	250	3,406
DMR				
HSA	35,763	13	6.46	16
BRA	1,043,116	20	268	14,909
SP				
HSA	25,421	13	6.26	12
BRA	394,765	21	158	12,446
PFP				
HSA	17,692	10	6.96	7
BRA	71,800	17	45	972

Table 6. *HSA*, *BRA* performance statistics.(*SG: Shape Graph*)

- PFP: Preflow-push maxflow algorithm [5]. This algorithm only updates labels of nodes and edges, and does not modify the graph structure.

4.1 Static Analysis Evaluation

Table 5 reports the results of static analysis of our benchmarks. We measure the size of benchmarks by the number of intermediate language (Jimple) instructions in the client program, excluding the code implementing the data structures accompanied by a specification. Columns 3 to 5 show the number of static optimization opportunities that our analysis enables. Galois protects application-specific objects (e.g., the cavity in DMR) using a variant of object-based STM, which can also benefit from our optimizations. Column 5 refers to those objects. In all cases, our analysis was precise enough to identify the maximum number of sites that were eligible for optimization, and it discovered the minimal set of latest failsafe points. The optimal result that we compare against was determined manually. Since our analysis is sound, we need to consider only the relatively few calls where the analysis does not suggest conflict detection or rollback logging optimizations.

4.1.1 Comparing Analyses: HSA vs. BRA

In Table 6, we compare our analysis using hierarchy summarization abstraction (*HSA*), with an analysis using backward reachability abstraction (*BRA*). The first column reports the total number of shape graphs (SG) explored by the analysis, which is a measure for the amount of work performed. We also report the average size of a shape graph, the average number of SG's per CFG location (our analysis uses roughly 1.43 CFG locations for a Jimple instruction) at the fixed point, and the running time of the analysis.

As expected, *HSA* generates a constant number of SG's at each program location, whereas in *BRA* the number of SG's increases as the benchmarks become more complex (from 45 SG's for PFP to 268 for DMR). The benefits of *HSA* are more striking as the complexity of the benchmark increases. For PFP, *BRA* generates

roughly 6 times more SG's than *HSA*, per CFG location. For DMR, in which the number of collections increases, *BRA* produces 41 times more structures. Additionally, we observe that in *BRA* we have more refined and, consequently, larger SG's. For all benchmarks the average SG size in *BRA* is roughly 1.6 times larger than in *HSA*. These facts lead to a significant state space explosion, which translates to increased work performed by *BRA* (for DMR we see a 29-fold increase in the number of generated SG's), and to increased running times. Thus, *HSA* is as precise as *BRA* but more efficient.

4.2 Experimental Evaluation of Optimizations

This section provides detailed performance results for each benchmark. To evaluate the performance gains obtained by different levels of sophistication of the analysis, we considered the following variants for each benchmark.

- O1: Baseline version: accesses within parallel loops to *all* objects are protected.
- O2: Iteration-private objects are not protected.
- O3: O2+ dominated shared objects are not protected.
- O4: O3+ duplicate lock acquisitions and unnecessary undo operations are eliminated.

Even in the baseline version, we do not protect object accesses made outside of parallel loops since the analysis required to enable this is trivial. At level O2, iteration-private objects are identified and accesses to them are not protected; this optimization by itself can be accomplished by a combination of flow-insensitive points-to and escape analysis. Optimization levels O3 and O4 target shared data; for these levels, a shape analysis similar to ours is necessary.

We performed our experiments using the Galois runtime system and a Sun Fire X2270 Nehalem server running Ubuntu Linux version 8.04. The system contains two quad-core 2.93 GHz Intel Xeon processors, which share 24 GB of main memory. We used the Sun HotSpot 64-bit server JVM, version 1.6.0. Each variant was executed nine times in the same instance of the JVM. We drop the first two iterations to account for the overheads of JIT compilation, and report results for the run with the median running time.

Because of the don't-care non-determinism of unordered-set iterators, different executions of the same benchmark/input combination may perform different numbers of iterations. Since our optimizations focus on reducing the overhead of each iteration and not on controlling the total number of iterations, we focus on a performance metric called *throughput*, which is the number of committed iterations per millisecond. For completeness, we also present other measurements such as the total running time, the number of committed iterations, the abort ratio, etc. Table 7 shows detailed results for all benchmarks.

4.2.1 Boruvka's Algorithm

We do not provide results for level O2, since the number of iteration private objects is insignificant. The number of committed iterations is exactly the same across all thread counts (this is a natural property of the algorithm since each committed iteration adds one edge to the MST). The analysis is successful in reducing the number of locks per iteration, and it correctly infers that the operator implementation is cautious.

The Boruvka algorithm takes roughly 141 seconds to run if we use 1 thread and optimization level O1, and 75 seconds if we use 8 threads and optimization level O4. At optimization level O4, no undo's are logged and the number of acquired locks in each iteration is substantially reduced. However, overall speedup is limited by the high abort ratio (for example, for 8 threads, the abort ratio is between 68% and 75% for all levels of optimization). The abort ratio decreases as the optimization level increases because if the time to execute an iteration is reduced, the iteration holds its locks for a smaller amount of time, reducing the likelihood of conflicts. This high abort ratio is intrinsic to the algorithm. The MST is built bottom-up, so towards the end of the execution, only the top few levels of the tree remain to be built and there is not much parallel work.

A Non-Cautious Boruvka Implementation. As we discussed in Section 2, removing the call to *getNeighbors* at `L3` results in non-cautious iterations. Our analysis successfully infers that the failsafe point along this program path moves from `L3` to `L5`. The only difference in the inferred method flags is in `L4`, where the call to `removeEdge` requires the `UNDO` flag instead of `NONE`. This example shows the utility of our analysis for optimizing programs in which the operator implementation is not cautious.

4.2.2 Delaunay Mesh Refinement

The number of committed iterations for this application is fairly stable across thread counts and optimization levels. Lock acquisitions drop dramatically in going from O2 to O3. The analysis deduces correctly that the operator implementation is cautious, which is why the number of undo's per iteration drops to zero at optimization level O4 (the number of undo's per iteration is stable in going from O2 to O3 because the re-triangulated cavity is constructed in private storage and then stored into the shared graph). The abort ratio is very small even for 8 threads.

The reductions in the average number of acquired locks and logged undo's per iteration are reflected directly in the running time. DMR takes 171 sec. to run if we use 1 thread and optimization level O1, and only 5 sec. if we use 8 threads and optimization level O4. This is roughly a factor of 34 improvement in the running time, of which a factor of roughly 8 comes from optimizations and a scaling factor of roughly 4 comes from increasing the number of threads. Since the number of committed iterations is fairly stable across all optimization levels and thread counts, the same improvement factors can also be seen in throughput.

4.2.3 Survey Propagation

The number of committed iterations is fairly stable for this benchmark. The analysis is successful in reducing the number of locks per iteration. The number of undo's per iteration is fairly small even at optimization level O1 because the graph is mutated only when a variable is frozen, which happens in very few iterations. The analysis correctly infers that the operator implementation is cautious.

The SP algorithm takes roughly 180 seconds to run if we use 1 thread and optimization level O1, and 9 seconds if we use 8 threads and optimization level O4. Most of this benefit comes from the optimizations; at optimization level O4, we observe a speedup of roughly 1.6 on 8 threads. We see a $5.5\times$ improvement in throughput for 8 threads when the optimization level goes from O1 to O4, and by 19% from O3 to O4.

4.2.4 Preflow-push Maximal Flow

A distinctive characteristic of PFP is its schedule sensitivity - because of don't-care non-determinism, different schedules can perform very different amounts of work. This can be seen in the 8-thread numbers: at optimization level O4, the program executes twice as many iterations on 8 threads as it does on a single thread. The number of undos per iteration is 0 for O3, since the graph structure is not mutated by the algorithm.

The preflow-push algorithm takes roughly 104 seconds to run if we use 1 thread and optimization level O1, and the best parallel time is 6.6 seconds if we use 4 threads and optimization level O4. This is a 16-fold improvement, of which roughly 6-fold im-

Th.	BVK			DMR				SP				PFP			
	O1	O3	O4	O1	O2	O3	O4	O1	O2	O3	O4	O1	O2	O3	O4
Lock Acquisitions/Iteration															
1	885	637	64	1,429	1,269	97	28	99	98	49	6	109	111	44	8
2	1,114	799	93	1,429	1,269	97	28	99	99	50	6	109	111	44	8
4	1,270	896	118	1,429	1,269	97	28	99	100	50	6	109	111	44	8
8	1,512	1,020	153	1,430	1,268	97	28	100	100	50	6	110	111	45	9
Undos/Iteration															
1	40.69	40.77	0.00	27.77	8.45	8.45	0.00	4.82	4.82	0.00	0.00	5.85	2.93	0.00	0.00
2	47.42	47.30	0.00	27.77	8.45	8.44	0.00	4.85	4.85	≈ 0.00	0.00	5.85	2.93	0.00	0.00
4	47.46	47.72	0.00	27.77	8.45	8.44	0.00	4.86	4.94	≈ 0.00	0.00	5.85	2.94	0.00	0.00
8	46.61	47.25	0.00	27.79	8.44	8.45	0.00	4.90	4.89	≈ 0.00	0.00	5.92	2.95	0.00	0.00
Committed Iterations (Millions)															
1	1.60	1.60	1.60	1.62	1.62	1.62	1.62	21.61	21.52	21.49	21.71	8.62	8.34	8.41	8.41
2	1.60	1.60	1.60	1.61	1.62	1.62	1.62	21.52	22.15	22.25	21.70	8.48	8.41	8.39	8.38
4	1.60	1.60	1.60	1.62	1.62	1.62	1.62	21.75	21.68	23.84	23.15	9.33	8.52	8.50	9.25
8	1.60	1.60	1.60	1.61	1.62	1.62	1.62	21.85	23.79	24.24	25.12	9.80	11.23	13.54	16.39
Abort Ratio %															
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	42.50	40.28	36.99	0.00	0.01	0.00	0.01	0.97	0.95	1.04	1.23	0.13	0.14	0.11	0.11
4	61.52	58.57	55.18	0.03	0.02	0.01	0.02	3.51	3.60	3.50	3.45	0.61	0.35	0.38	0.87
8	75.70	71.25	68.57	0.05	0.04	0.02	0.03	8.39	8.62	11.54	7.19	1.02	3.08	5.04	3.83
Running Time (sec)															
1	141	107	81	171	132	24	22	180	176	24	14	104	94	22	18
2	155	109	81	93	77	14	12	109	110	24	17	75	69	13	10
4	190	102	77	49	39	8	7	65	58	14	11	114	103	7.3	6.6
8	219	98	75	28	22	5	5	42	42	10	9	108	108	7.7	9.4
Throughput (Iterations/ms)															
1	11.38	15.01	19.65	9	12	68	75	120	122	893	1,533	83	89	389	463
2	10.34	14.71	19.72	17	21	118	131	198	201	934	1,274	113	122	659	798
4	8.41	15.62	20.87	33	41	212	234	333	371	1,729	2,074	82	83	1,167	1,402
8	7.31	16.38	21.40	57	73	323	347	515	573	2,404	2,868	91	104	1,762	1,739

Table 7. Performance metrics. BVK input is a random graph of 800,000 nodes and 5-10 neighbors per node. DMR input is a random mesh with 549,998 total triangles, 261,100 bad. SP input is a 3-SAT formula with $1,000$ variables and $4,200$ clauses. PFP input is a random graph of $262,144$ nodes and capacities in the range [0, 10000].

provement comes from the optimizations, and an improvement of roughly 3-fold comes from exploiting parallelism.

4.2.5 Summary of Results

Our analysis eliminates all costs related to rollback logging for our benchmarks, and reduces the number of lock acquisitions by a factor ranging from $10\times$ to $50\times$, depending on the application and the number of threads. These improvements translate to a noticeable improvement (ranging from $2\times$ up to $12\times$) in the running time, which is consistent across different thread counts, and robust against pathologies of speculation (e.g. high abort ratio).

5. Related Work

Prior work on shape analysis has focused mostly on analyzing data structure implementations to infer heap structure. In contrast, we use data structure specifications to abstract away data structure representations, and we focus on unstructured graphs.

The Jahob system [19] verifies that a data structure implementation meets its specification, and it uses the abstract state to simplify the verification of data structure clients. Our analysis assumes that a given specification is correct. Checking that the implementation and specification of the method semantics match and that the `@locks` specification ensures that only commutating methods can execute concurrently is an interesting challenge.

Maron et al. [24] use specialized predicates to model sharing patterns between objects stored in data structures, and use this information to statically parallelize benchmarks from the JOlden suite and SPECjvm98 benchmarks. Our benchmarks operate on unstructured graphs and are not amenable to static parallelization. We exploit the fact that our execution model is speculative to avoid tracking correlations between different data structures, which increases the cost of the analysis considerably.

In the current Galois system, the optimizations described here are performed manually [26]. Our shape analysis automates these optimizations, reducing the burden on the programmer and ensuring correctness of optimized code. Failsafe points extend the notion of cautious operators. Our running example shows that non-cautious code too can be optimized by turning conflict detection and rollback logging off for a subset of the calls, obtaining performance improvement similar to the cautious version. Additionally, in [26] the system optimizes locking only after the failsafe point in contrast to our analysis, which optimizes locking regardless of whether an operator is cautious.

Prabhu et al. [29] use value speculation to probabilistically reduce the critical path length in ordered algorithms. Their static analysis focuses mainly on array programs. Value speculation is orthogonal to our approach, and the benchmarks discussed in this paper do not benefit from value speculation. Furthermore, our heap abstractions are very different because we need to handle complex ADTs such as unstructured graphs.

Harris et al. [10], Adl-Tabatabai et al. [1], and Dragojevic et al. [6] use compiler optimizations to reduce the overheads of transactional memory. They also handle immutable, and transaction local objects. Additionally, they describe extending traditional compiler optimizations such as common subexpression elimination (CSE) to reduce the overheads of logging. Although CSE helps to reduce repeated logging for a single object, its effectiveness for our benchmarks is limited by the extensive use of collections. Their

approaches cannot capture global properties such as failsafe points. Other optimizations they propose are complementary to ours.

McCloskey et al. [25], Hicks et al. [13], and Cherem et al. [4] describe analyses that infer locks for atomic sections. These techniques are overly conservative for our benchmarks since they would always infer that an iteration might touch the whole graph.

Acknowledgments

We would like to thank the anonymous referees, Noam Rinetzky, and Josh Berdine for their helpful comments.

References

[1] A. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI*. ACM, 2006.

[2] A. Braunstein, M. Mèzard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms*, 27(2):201–226, 2005.

[3] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Footprint analysis: A shape analysis that discovers preconditions. In *SAS*, 2007.

[4] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI*. ACM, 2008.

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, editors. *Introduction to Algorithms*. MIT Press, 2001.

[6] A. Dragojevic, Y. Ni, and A. Adl-Tabatabai. Optimizing transactions for captured memory. In *SPAA*, 2009.

[7] D. Eppstein. *Spanning trees and spanners*, pages 425–461. Elsevier, 1999.

[8] I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *ESOP*, 2009.

[9] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03*, 2003.

[10] T. L. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI*. ACM, 2006.

[11] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPOPP*. ACM, 2008.

[12] M. Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.

[13] M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *TRANSACT*, June 2006.

[14] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[15] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*, 2009.

[16] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *SPAA '08*, 2008.

[17] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*. ACM, 2007.

[18] V. Kuncak and M. C. Rinard. Decision procedures for set-valued fields. *Electr. Notes Theor. Comput. Sci.*, 131, 2005.

[19] V. Kuncak and M. C. Rinard. An overview of the jahob analysis system: project goals and current status. In *IPDPS*, 2006.

[20] S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Log.*, 9(1), 2007.

[21] P. Lam, V. Kuncak, and M. C. Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Notices*, 39(3), 2004.

[22] T. Lev-Ami and M. Sagiv. TVLA: A framework for implementing static analyses. In *SAS*, 2000.

[23] R. Manevich, S. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *SAS*, 2004.

[24] M. Marron, D. Stefanovic, D. Kapur, and M. V. Hermenegildo. Identification of heap-carried data dependence via explicit store heap models. In *LCPC*, pages 94–108, 2008.

[25] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*. ACM, 2006.

[26] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *PPOPP*. ACM, 2010.

[27] K. Pingali, M. Kulkarni, D. Nguyen, M. Burtscher, M. Mendez-Lojo, D. Prountzos, X. Sui, and Z. Zhong. Amorphous data-parallelism in irregular algorithms. regular tech report TR-09-05, The University of Texas at Austin, 2009.

[28] A. Podelski and T. Wies. Boolean heaps. In *SAS*, 2005.

[29] P. Prabhu, G. Ramalingam, and K. Vaswani. Safe programmable speculative parallelism. In *PLDI*, 2010.

[30] D. Prountzos, R. Manevich, K. Pingali, and K. S. McKinley. A shape analysis for optimizing parallel graph programs. Technical Report TR-10-27, UT Austin, http://userweb.cs.utexas.edu/users/dprountz/UTCS-TR-10-27.pdf, Jul 2010.

[31] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel Distrib. Syst.*, 10(2):160–180, 1999.

[32] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3), 2002.

[33] A. Salcianu and M. C. Rinard. Purity and side effect analysis for java programs. In *VMCAI*, 2005.

[34] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *PPOPP*. ACM, 2008.