HAWKEYE: Effective Discovery of Dataflow Impediments to Parallelization

Omer Tripp *
Tel Aviv University
omertrip@post.tau.ac.il

Greta Yorsh †
ARM
greta.yorsh@arm.com

John Field †
Google
jfield@google.com

Mooly Sagiv
Tel Aviv University
msagiv@post.tau.ac.il

Abstract

Parallelization transformations are an important vehicle for improving the performance and scalability of a software system. Utilizing concurrency requires that the developer first identify a suitable parallelization scope: one that poses as a performance bottleneck, and at the same time, exhibits considerable available parallelism. However, having identified a candidate scope, the developer still needs to ensure the correctness of the transformation. This is a difficult undertaking, where a major source of complication lies in tracking down sequential dependencies that inhibit parallelization and addressing them.

We report on HAWKEYE, a dynamic dependence-analysis tool that is designed to assist programmers in pinpointing such impediments to parallelization. In contrast with field-based dependence analyses, which track concrete memory conflicts and thus suffer from a high rate of false reports, HAWKEYE tracks dependencies induced by the abstract semantics of the data type while ignoring dependencing arising solely from implementation artifacts. This enables a more concise report, where the reported dependencies are more likely to be real as well as intelligible to the programmer.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Measurement,Experimentation

Keywords loop parallelization, commutativity, dependence analysis, dynamic analysis, abstract data types

1. Introduction

Program dependence graphs are a useful aid in software parallelization [13]. However, parallelization of existing programs using dependence analysis remains challenging. One reason is the inherent difficulty of computing static dependencies in programs with complex data structures. Yet even the *dynamic* dependencies exhibited by a single execution trace may embody superfluous information.

Consider the following illustrative example (in Java):

```
1 Pair[] pairs = makePairs();
2 Map m = new ConcurrentHashMap();
3 for (int i=0; i<pairs.length; ++i) {
4   m.put(pairs[i].fst, pairs[i].snd);
5   compute(pairs[i].snd); }
```

Assume that (i) the `fst` field of each `Pair` object points to a unique object, but there are two distinct `Pair` objects whose `snd` fields point to a shared object, and (ii) `compute` modifies some fields of the object pointed-to by its argument.

A dynamic dependence-analysis tool based on field reads and writes will infer that the two relevant `compute` invocations depend on each other, which would in turn inhibit parallelization of the loop. However, it would also indicate dependencies between `put` calls. These dependencies arise from access to the internal fields of `ConcurrentHashMap`.[1] These dependencies are spurious, in the sense that they do not reflect genuine constraints on parallelization.

Spurious dependencies obscure the real constraints on parallelization, and thus stand in the developer's way toward

* Research supported by an Open Cooperative Research grant from IBM Research to Stanford and Tel Aviv Universities.

† Research done when author was a staff member at IBM Research.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

[1] For example, when we ran this code using the IBM V9 JRE, dependencies were reported on `ConcurrentHashMap$HashEntry.hash`, `ConcurrentHashMap$Segment.modCount`, and several other internal fields of `ConcurrentHashMap`.

arriving at a parallel version of the code. In the above example, instead of focusing the developer's attention on the only real impediment to parallelization—the two `compute` calls operating on the same object—a concrete dependence analysis would also overwhelm the developer with (potentially many) `put`/`put` conflicts that can all safely be ignored.

Scope This paper takes a step toward reporting useful dynamic dependencies in support of programmer-guided parallelization. We assume that the programmer has already identified a suitable scope for parallelization, where there are both performance bottlenecks and a great deal of available parallelism. The remaining challenge is to uncover (the few) dependencies that impede parallelization and address them.

Naïve reporting of (concrete) data dependencies often yields a prohibitive amount of spurious dependencies. It is then the developer's responsibility to distill the real constraints on parallelization, a task that in many cases obviates the value of using a dependence analysis in the first place. This paper tackles the challenge of reporting dataflow impediments to parallelization in a precise, concise and intelligible manner.

Our Approach We build on the centrality of abstract data types (ADTs) in the design and development of object-oriented software. Data structures implementing ADTs can come from libraries, but may also be user-defined data types.

In our approach, concrete data structures are considered in terms of the semantics of the ADTs they represent during dependence analysis. This allows the analysis to suppress spurious dependencies due to the specifics of the ADT implementation, and report conflicts at the semantic level. We have implemented our approach in HAWKEYE, a dynamic dependence analysis that accounts for ADT semantics.

Past research has already addressed the problem of spurious dependencies, though there are some important differences, which we now briefly discuss. The commutativity analysis framework [30–32], used in parallelizing compilers, employs symbolic reasoning and other specialized algorithms to recognize and exploit commuting operations. HAWKEYE, instead, is a dynamic analysis. HAWKEYE also utilizes abstraction, rather than commutativity proofs, to ignore spurious dependencies.

HAWKEYE is also reminiscent of recent works in the area of transactional memory [16, 20–23] that leverage ADT semantics for more robust online conflict detection. Unlike these works, HAWKEYE performs offline conflict detection. Moreover, the HAWKEYE specification is in the form of a representation function, rather than a commutativity specification, which—in our experience—facilitates support for user types.

Contributions This paper makes the following contributions:

- ***Effective dependence analysis.*** Our approach augments dynamic dependence analysis with conflict detection based on ADT semantics. This supports our objective of concise and precise dependencies, as our experimental results confirm. The reported dependencies are also more intelligible to the user being at the semantic level.
- ***Flexible specification language.*** HAWKEYE enables a wide spectrum of specifications for a given ADT that represent different tradeoffs between the complexity of the specification and the cost and precision of the analysis. Our experience suggests that the simplest form of specification, whereby commutativity between ADT operations is inferred automatically based (only) on a definition of the ADT's representation function, suffices for most ADTs. This facilitates the definition and incorporation of user types into the analysis.
- ***Uniform conflict-detection framework.*** The theoretical underpinnings of our analysis, which we evolve in Sections 3–6, enable uniform treatment of concrete and semantic dependencies. This property of our framework simplifies reasoning on the behavior of the analysis, as well as implementation effort.
- ***Implementation and evaluation.*** We have implemented our approach in HAWKEYE, and evaluated it via two sets of experiments. First, we compared HAWKEYE and an analogous analysis that is unaware of ADT semantics by using both to discover loop-carried dependencies in seven real-world benchmarks. We then applied parallelization transformations in three of these benchmarks guided by the dependencies reported by HAWKEYE to assess its value in end-to-end parallelization. The results are highly encouraging: HAWKEYE reported significantly fewer dependencies than the baseline analysis, and the surviving dependencies represented real impediments to parallelization, which greatly facilitated manual parallelization of the benchmarks we studied.

Organization In the remainder of this paper, we first illustrate our approach via a real-world example in Section 2. Then, in Sections 3–6, we establish a uniform framework for detecting both semantic conflicts between ADT operations and concrete conflicts between non-ADT operations. The implementation of HAWKEYE is described in Section 7, which also reports on its experimental evaluation. Section 8 discusses related work. We conclude in Section 9.

2. Overview

Consider the (pseudo-)code fragment in Fig. 1 taken from the JGraphT library [4], which builds the block-cutpoint graph [33] representation of a connected undirected graph. The block-cutpoint graph of connected undirected graph G is a bipartite graph connecting "cutpoints" in G to their containing "blocks", where node v in G is called a *cutpoint* if its removal disconnects G, and subgraph b of G is considered a *block* if it is a maximal subgraph of G not containing a cutpoint as an independent graph (i.e., disregarding the rest of G). Fig. 2 presents a simple example of a connected graph (G) and its corresponding block-cutpoint graph (G').

```
1  for (Vertex cutpoint : this.cutpoints) {
2    UndirectedGraph subgraph = new SimpleGraph();
3    subgraph.addVertex(cutpoint);
4    this.cutpointGraphs.put(cutpoint, subgraph);
5    this.addVertex(subgraph);
6    Set blocks = this.vertex2blocks.get(cutpoint);
7    for (UndirectedGraph block : blocks) {
8      int oldHitCount = this.block2hits.get(block);
9      this.block2hits.put(block, oldHitCount+1);
10     this.addEdge(subgraph, block); } }
```

Figure 1. Simplified pseudo-code version of the JGraphT algorithm for building a block-cutpoint graph

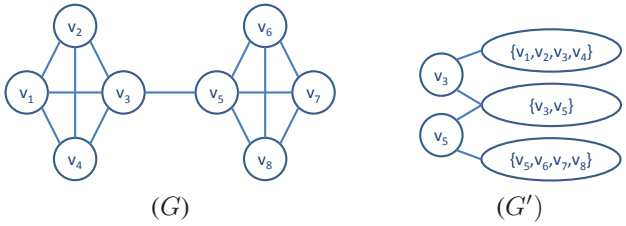

Figure 2. Connected undirected graph G and its corresponding block-cutpoint graph G'

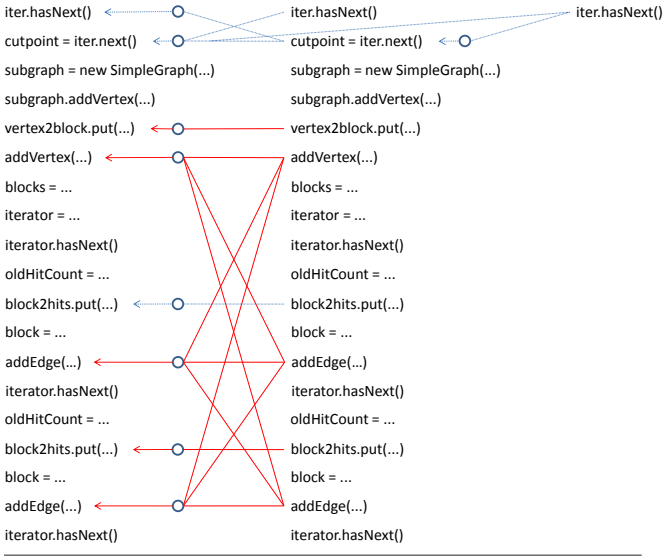

Figure 3. Dynamic dependence graph for the program in Fig. 1 running on the graph in Fig. 2

The code in Fig. 1 builds a `BlockCutpointGraph` instance, G' (pointed-to by the `this` pointer), where the nodes of G' are themselves graphs, by iterating over the cutpoints in the graph G underlying G'. For each cutpoint c, (i) a fresh graph enclosing c (line 3) is added to G (line 5), (ii) the blocks c is contained in (which were computed ahead of the loop) are retrieved (line 6), and (iii) these blocks are traversed by an inner loop (line 7) that inserts an edge into G' between c and each of them via an `addEdge` call (line 10). Prior to the edge insertion, a counter counting the number of times each block was encountered is incremented (line 9).

Program Parallelization The loop in Fig. 1 has a great deal of available parallelism: Different iterations process distinct cutpoints, and thus manipulate disjoint portions of the block-cutpoint graph. Dependence analysis is a useful technique for testing this observation: Loop-carried dependencies reported by the analysis pose as potential impediments to parallelization, which the developer can review and — if needed — address before parallelizing the loop. Dually, failure to find dependencies (by a sound analysis) is a correctness proof for the parallelization transformation.

Unfortunately, existing dependence analyses, which track conflicts at the level of conrete memory locations, are overly pessimistic. Fig. 3, which visualizes the dynamic loop-carried dependencies recorded when running the loop with graph G from Fig. 2 as input, illustrates this. The many reported dependencies obscure the parallelization potential latent in the loop. Most of these dependencies are due to the internals of collection implementations (e.g., the `HashMap` implementing `vertex2block`), and are thus spurious. Parallelizing compilers are likely to arrive at an even more conservative result due to their limited ability to handle aliasing and track graph invariants maintained by G (which is a tree).

The important dependencies (denoted in red) are (i) those involving the induction variable (`iter`), as well as (ii) a specific pair of calls to `put` on `block2hits` where block $\{v_3, v_5\}$, which is shared by cutpoints v_3 and v_5, is used as the key. Each of these essential dependencies can be addressed by standard parallelization techniques. For (i), *cutpoints* can be changed into a random-access list, or alternatively, the sequential loop structure can be preserved while making the loop body asynchronous. For (ii), atomicity can be guaranteed for the statements at lines 8–9 via appropriate synchronization (e.g., locks or transactions), or `block2hits` can be privatized and its different copies can be merged after the loop.

As this example demonstrates, often the challenge in parallelization is not *how* to transform the code, but *where* to apply the transformation. A developer confronted with the dependence graph of Fig. 3 is likely to spend a long time separating the wheat from the chaff, or even give up at some point. Our goal is to distill the report by a dependence analysis toward more concise and precise reporting of candidate impediments to parallelization.

Dependence Analysis with ADTs In our approach, ADT instances are treated at the semantic rather than the concrete level. The developer specifies the ADT implemented by a concrete type (that is known to have a concurrent implementation), and the analysis then computes dependencies according to this specification, which enforces a more precise notion of commutativity between ADT operations. The result by the analysis is then more accurate and thus also more actionable.

Representation Functions for `Map` and `UndirectedGraph`

MAP(m):
  $state = \varnothing$
  for each e in $m.entrySet$
    $state = state \cup m \overset{key(e)}{\rightarrow} e.key$
    $state = state \cup e.key \overset{value(e)}{\rightarrow} e.value$
  return $state$

UNDIRECTEDGRAPH(g):
  $state = \varnothing$
  for each n in $g.nodeSet$
    $state = state \cup g \overset{node(n)}{\rightarrow} n$
  for each e in $g.edgeSet$
    $state = state \cup \{e.n_1 \overset{target(e)}{\rightarrow} e.n_2, e.n_2 \overset{target(e)}{\rightarrow} e.n_1\}$
  return $state$

Figure 4. Pseudo-code version of the representation functions for the `Map` and `UndirectedGraph` ADTs used by the program in Fig. 1

Our choice of description for ADTs is as a store mapping logical fields to their respective values. For example, the `Map` ADT can represent keys as logical fields that point to their associated values, which is the natural way of perceiving a mapping. A pleasing property of this representation is that it enables a uniform conflict-detection framework based on data dependencies, which we formalize in Sections 3–6.

HAWKEYE accepts ADT specifications in the form of a Java method receiving a concrete data-structure instance as input and returning the object's state as an ADT. This is illustrated in Fig. 4, where — for example — the specification for `UndirectedGraph` lets the state of graph instance g map g to the nodes it contains via entries of the form $\langle g, node(n)\rangle \rightarrow n$, and node u map to its neighbors via entries of the form $\langle u, target(e)\rangle \rightarrow v$ (where $e = \{u, v\}$). For the program in Fig. 1, HAWKEYE reports all the essential dependencies (those in red), and only these dependencies, based only on a user-provided representation function for the `UndirectedGraph` ADT. (HAWKEYE already has built-in representation functions for the Java collections.)

Technical Outline The algorithm undelying HAWKEYE is evolved incrementally in Sections 3–6. Sections 3–4 formalize the connection between parallelization and dynamic data dependencies. First, in Section 3, we define a correctness criterion for parallel execution of a single trace (assuming an interleaving semantics of concurrency) whereby a parallel schedule is correct iff all the interleavings it permits of statements from its underlying trace yield the final state of the trace. We further discuss how (local) reasoning about commutativity between adjacent trace transitions can be utilized toward conservative enforcement of this criterion. Section 4 then links the criterion to data dependencies based on the observation that absence of data dependencies implies commutativity. Importantly, this section defines the read and write accesses made by a trace transition in a way that we can later use both in concrete and in abstract semantics, which is key for the uniformity of our framework.

Sections 5–6 build on the results of the two preceding sections. Section 5 instantiates the definitions of Section 4 both in a standard concrete semantics where a state is encoded as a store mapping stack and heap locations to their corresponding values, and in an abstract semantics where states are structured as mappings from abstract locations to abstract values. Finally, Section 6 considers abstraction using representation functions, and presents an efficient algorithm for computing dependencies between ADT operations based on the definitions of Section 4.

3. Correct Parallelization

We start by introducing the notations we use for program states and traces. We then discuss how commutativity between statements in a trace can be used to reason about the correctness of its parallel execution.

Low-level Program Semantics A state $\sigma\colon L \hookrightarrow V$ is a mapping from memory locations to values. We use $dom(\sigma)$ to denote the domain of the mapping, i.e., the subset of memory locations for which the partial function σ is defined. The intended meaning is that $dom(\sigma)$ includes the memory locations that are allocated in state σ (see Section 5.1). The set of all states is denoted by Σ. We use $[\![p]\!](\sigma)$ to denote the state resulting from executing program statement p in state $\sigma \in \Sigma$. A transition τ is a triple $\langle \sigma, p, \sigma'\rangle$, where p is a program statement and $\sigma, \sigma' \in \Sigma$ are states such that $\sigma' = [\![p]\!](\sigma)$. The set of all transitions is denoted by T. A trace t is a sequence $\langle \tau_1, \tau_2, \ldots, \tau_n\rangle$ of transitions, such that $\tau_i = \langle \sigma_i, p_i, \sigma_i'\rangle \in T$ for $1 \leq i \leq n$ and $\sigma_i' = \sigma_{i+1}$ for $1 \leq i < n$.

Parallel Execution Semantics Let $t = \langle \tau_1, \tau_2, \ldots, \tau_n\rangle$ be a trace, where p_i is the i-th statement executed in t. Consider a partitioning of the statements p_i into consecutive segments $I_1, \ldots, I_k$ according to indices $1 = i_1 < \ldots < i_{k+1} = n + 1$, where $I_j = \langle p_{i_j}, \ldots, p_{i_{j+1}-1}\rangle$. I_j intuitively corresponds to a single command, which preserves the ordering between the statements comprising it under parallelization.

We define an execution schedule for $I_1, \ldots, I_k$ via the standard composition operators:

1. **Sequential composition.** We denote the sequential execution of I_j and I_{j+1} by $I_j; I_{j+1}$.
2. **Parallel composition.** We denote concurrent execution of consecutive segments I_j and I_{j+1} by $I_j \| I_{j+1}$. $\|$ is transitive in that $I_j \| I_{j+1} \| I_{j+2}$ denotes that I_j, I_{j+1} and I_{j+2} will be run concurrently to each other.

An execution schedule of segments $I_1, \ldots, I_k$ corresponding to trace t is thus of the form

$$I_1 *_1 I_2 *_2 \ldots *_{k-1} I_k, \tag{1}$$

where $*_m \in \{;, \|\}$ for $1 \leq m \leq k$.

For example, schedule $I_1; I_2 \| I_3 \| I_4; I_5$ prescribes that first I_1 is run, then I_2, I_3 and I_4 run concurrently to each other, and finally I_5 is executed.

We assume an interleaving semantics of concurrency [27, 28], where the single steps of concurrent segments are interleaved with each other (in an unscheduled way). For $I_j \| \ldots \| I_{j+m}$, all possible permutations of the statements comprising these segments that preserve the internal ordering of statements in $I_j, \ldots, I_{j+m}$ are possible.

Commutativity Let $t = \langle \tau_1, \tau_2, \ldots, \tau_n \rangle$ be a trace. We denote the set of statements executed in t by $P(t) = \{p_i \mid 1 \leq i \leq n\}$. We say that consecutive statements $p_i, p_{i+1} \in P(t)$ *commute* in t if the trace from running $\langle p_1, \ldots, p_{i+1}, p_i, \ldots, p_n \rangle$ starting at σ_1 has final state σ'_n. That is, swapping between p_i and p_{i+1} does not change the final state of the run.

We refer to two traces as equivalent if (i) they are both over the same set of statements, and (ii) their starting and final states are identical:

$$t = \langle \tau_1, \ldots, \tau_n \rangle \equiv \widehat{t} = \langle \widehat{\tau}_1, \ldots, \widehat{\tau}_n \rangle \Leftrightarrow \\ \sigma_1 = \widehat{\sigma}_1 \wedge P(t) = P(\widehat{t}) \wedge \sigma'_n = \widehat{\sigma'}_n. \quad (2)$$

Our definition of commutativity between statements supports this notion of equivalence. Denote by $\vec{P}(t)$ the organization of the statements in $P(t)$ as a sequence (according to their ordering in t). Considering traces t and $\widehat{t}$, if (i) $\sigma_1 = \widehat{\sigma}_1$, and (ii) $\vec{P}(\widehat{t})$ is a transposition of $\vec{P}(t)$ where the transposed statements are commutative in t, then by definition t and $\widehat{t}$ are equivalent according to (2). We record this connection between t and $\widehat{t}$ via (symmetric) relation $\equiv_1$: $t \equiv_1 \widehat{t}$. We can now define $\equiv_n$ inductively:

$$t \equiv_n t' \Leftrightarrow \exists t''. t \equiv_{n-1} t'' \wedge t'' \equiv_1 t'.$$

Note that for all $n \in \mathbb{N}$, if $t \equiv_n t'$ then $t \equiv t'$.

Correct Parallel Schedule Let t be a trace and S a (parallel) schedule based on t (according to (1)). We say that S is *correct* with respect to t if every possible trace t' resulting from executing S is equivalent to t. If we consider t as a single run of some program P, then a complementary perspective is enabled: A transformation of P that makes P more concurrent is incorrect if it results in a parallel execution schedule based on t yielding traces that are inequivalent to t. In this sense, information from a single trace provides an upper bound on available parallelism that can be used to test a proposed transformation.

Note that we can use any subset of the equivalence relation defined in (2) toward a conservative judgment about the correctness of a schedule, which may reject valid schedules being unable to prove them as such. In particular, we can use $\bigcup_{k \in \mathbb{N}} \equiv_k \subseteq \equiv$ or even a subset of $\bigcup_{k \in \mathbb{N}} \equiv_k$ that is built based on a safe approximation of the commutativity relation between statements, where a safe commutativity judgment is one where two statements are said to be commutative (in some trace) if they actually commute, but the opposite is not necessarily true.

The motivation for using a conservative correctness judgment stems from the cost and tractability of direct reasoning about equivalence between traces. In what follows, we approximate the commutativity property by considering data dependencies between trace transitions.

4. Formalizing Dependencies

In this section, we formally define the notion of data dependencies. We then restate the (well-known) connection between data dependencies and commutativity [13], whereby absence of data dependencies implies commutativity, in our setting of dynamic dependencies recorded based on a single execution trace.

4.1 Trace Dependence Graph

We first define the notion of best read and write sets, as well as and their conservative approximation. We then use these sets to define dependencies in a standard way.

Read and Write Sets Each transition $\tau = \langle \sigma, p, \sigma' \rangle$ is associated with a write set $write(\tau)$ and a read set $read(\tau)$ of memory locations, which we now define.

The write set is the set of memory locations modified by the transition:

$$write(\tau) \triangleq mod(\tau) \cup alloc(\tau) \cup dealloc(\tau) \quad (3)$$

where

$$\begin{aligned} mod(\tau) &\triangleq \{m \in dom(\sigma) \cap dom(\sigma') \mid \sigma(m) \neq \sigma'(m)\} \\ alloc(\tau) &\triangleq dom(\sigma') \setminus dom(\sigma) \\ dealloc(\tau) &\triangleq dom(\sigma) \setminus dom(\sigma') \end{aligned}$$

The first line handles change of values of existing locations. The second and third lines handle memory allocation and deallocation, respectively. We assume that identities of memory locations do not change, and that deallocation is an explicit transition (i.e., the garbage collector is treated as part of the program).

The read set is the set of memory locations whose values determine the effect of p. Intuitively, a memory location m is in the read set of $\langle \sigma, p, \sigma' \rangle$ if altering its value in σ affects the result of executing p. Formally, to ensure its uniqueness, the read set is defined as the union of all minimal sufficient read sets:

$$read(\tau) \triangleq \bigcup \{M \in R(\tau) \mid \forall M' \in R(\tau). M' \not\subset M\} \quad (4)$$

where $R(\tau)$ is the set of all sufficient read sets of transition τ, i.e., $M \in R(\tau)$ if and only if $M \subseteq dom(\sigma)$ and for every

$\widehat{\sigma}, \widehat{\sigma}' \in \Sigma$,

$$\begin{array}{lll} \text{if} & M \subseteq dom(\widehat{\sigma}) & \\ & \forall m \in M.\sigma(m) = \widehat{\sigma}(m) & \\ & \widehat{\tau} = \langle \widehat{\sigma}, p, \widehat{\sigma}' \rangle \in T & \\ \text{then} & mod(\tau) = mod(\widehat{\tau}) & (5) \\ & alloc(\tau) = alloc(\widehat{\tau}) & \\ & dealloc(\tau) = dealloc(\widehat{\tau}) & \\ & \forall m \in write(\tau) \cap dom(\sigma').\sigma'(m) = \widehat{\sigma}'(m) & \end{array}$$

Intuitively, M is a sufficient read set if whenever the states σ and $\widehat{\sigma}$ agree on the values of all memory locations in M (but may differ in other values), the results of applying p to them also agree on what locations change and how.

Dependencies There is a conflict between two transitions when one of them writes a memory location that is accessed (for read or write) by the other.

Let $r, w \colon T \to \mathcal{P}(L)$ define some read and write sets. The following definitions are (implicitly) parameterized by r and w. We use $cm(\tau, \widehat{\tau})$ to denote the set of locations that participate in a conflict between two transitions, τ and $\widehat{\tau}$:

$$cm(\tau, \widehat{\tau}) \triangleq w(\tau) \cap (w(\widehat{\tau}) \cup r(\widehat{\tau})) \ \cup \ w(\widehat{\tau}) \cap (w(\tau) \cup r(\tau)) \quad (6)$$

We say that there is a conflict between τ and $\widehat{\tau}$ (or τ and $\widehat{\tau}$ are conflicting) when $cm(\tau, \widehat{\tau}) \neq \varnothing$. The conflict relation is symmetric.

The set of dependencies of trace $t = \langle \tau_1, \ldots, \tau_n \rangle$, defined with respect to a given w and r, is the set of pairs of conflicting transitions of the trace t, ordered by time:

$$D[r, w](t) = \{\langle \tau_i, \tau_j \rangle \mid i < j, cm(\tau_i, \tau_j) \neq \varnothing\} \quad (7)$$

We use $\tau_i \leftarrow \tau_j$ to denote that the dependency $\langle \tau_i, \tau_j \rangle$ is in $D[r, w](t)$, when t, r, w are clear from the context. For simplicity, we do not distinguish between RAW, WAR, and WAW dependencies. It is easy to see that by using larger read and write sets, we get more dependencies. A weaker condition is stated in the following.

Lemma 4.1. *Let $r, w, r', w' \colon T \to \mathcal{P}(L)$ such that for all $\tau \in T$, $r(\tau) \subseteq r'(\tau) \cup w'(\tau)$ and $w(\tau) \subseteq w'(\tau)$. For every trace t, $D[r, w](t) \subseteq D[r', w'](t)$.*

Proof. See Appendix A. □

In particular, a location read by τ according to r can be in either $r'(\tau)$ or $w'(\tau)$. Mappings $r, w \colon T \to \mathcal{P}(L)$ define *conservative* read and write sets when for all $\tau \in T$,

$$read(\tau) \subseteq r(\tau) \cup w(\tau) \ \text{ and } \ write(\tau) \subseteq w(\tau) \quad (8)$$

When referring to the "best" read and write sets, we intend those defined in (4) and (3). Otherwise, when we mention read and write sets, we refer to conservative read and write sets as defined in (8). It follows from Lemma 4.1 that by using conservative read and write sets, we do not miss any dependencies that arise using the best read and write sets.

Trace Dependence Graph In the rest of this paper, we refer to the (directed acyclic) graph induced by $D[r, w](t)$ as a "trace dependence graph" of t.

The nodes of this graph are the transitions of t, and the edges are the dependencies of $D[r, w](t)$.[2] The trace dependence graph of t is denoted by $G[r, w](t)$. We write $D(t)$ and $G(t)$ instead of $D[r, w](t)$ and $G[r, w](t)$, respectively, when the parameters r, w are clear from the context. We lift the definition of $G(t)$ to a (possibly infinite) set of traces by taking the union of the individual trace dependence graphs in the set.

4.2 Connection to Commutativity

Our goal in recording data dependencies is to enable reasoning about correct parallelization. We must therefore clarify the connection between dependencies and commutativity to be able to use the criterion specified in Section 3. The following establishes this connection.

Lemma 4.2. *Consider trace $t = \langle \tau_1, \ldots, \tau_n \rangle$, and let $D[r, w](t)$ denote the set of dependencies of t according to conservative read and write sets r and w. Then*

1. *every pair $\langle p_i, p_{i+1} \rangle$ of consecutive statements in $\vec{P}(t)$ such that $\langle \tau_i, \tau_{i+1} \rangle \notin D[r, w](t)$ is commutative in t.*
2. *Moreover, for every pair t and $\widehat{t}$ of traces and permutation δ, such that (i) $\vec{P}(\widehat{t}) = \delta(\vec{P}(t))$, (ii) $\widehat{\sigma}_1 = \sigma_1$, and (iii) for all $\langle \tau_i, \tau_j \rangle \in D[r, w](t)$, $\delta(i) < \delta(j)$, there exists a natural number $m \in \mathbb{N}$ such that $t \equiv_m \widehat{t}$.*

Proof. See Appendix B. □

The second clause of the claim states that if trace $\widehat{t}$ is the result of running the statements in t in a different order that does not break any data dependency, then the two traces are equivalent according to (2). Furthermore, we can arrive at $\vec{P}(\widehat{t})$ starting from $\vec{P}(t)$ via a finite number of "permitted" transpositions (according to the conservative notion of commutativity imposed by r and w).

This observation enables simple reasoning about the correctness of (parallel) schedule S based on trace t: As long as S enables only traces that satisfy the second claim above, and in particular, every possible trace of S satisfies the constraints imposed by $D[r, w](t)$, S is correct.

5. Concrete and Abstract Dependencies

We now explore the meaning of dependencies under abstraction. We show how the general definition of dependencies we gave in the previous section can be used with both concrete and abstract semantics, including specialized abstractions for ADTs and loop parallelization. Fig. 5 summarizes the notations for these semantic domains.

[2] Note the difference from standard program dependence graphs, where nodes are program statements. Also, our presentation is in terms of data dependencies only, without loss of generality, because control dependencies are subsumed by trace order.

Low-level Semantics	
$\sigma\colon L \hookrightarrow V$	$\sigma \in \Sigma$
$\tau \triangleq \langle \sigma, p, \sigma' \rangle$	$\tau \in T$
Simple Concrete Semantics	
$Val \triangleq Obj \uplus \{\texttt{null}\}$	
$\rho\colon \texttt{VarId} \hookrightarrow Val$	$\rho \in Env$
$heap\colon Obj \times \texttt{FieldId} \hookrightarrow Val$	$heap \in Heaps$
$state = \langle \rho, heap \rangle$	$state \in States$
$L \triangleq (Obj \times \texttt{FieldId}) \uplus VarId$	
$V \triangleq Val$	
$\sigma = heap \uplus \rho$	
Concrete Semantics with Methods	
$Val \triangleq Obj \uplus \{\texttt{null}\}$	
$\rho\colon \texttt{VarId} \hookrightarrow Val$	$\rho \in Env$
$stack \triangleq \langle \rho_1, \ldots, \rho_m \rangle$	$stack \in Stacks$
$heap\colon Obj \times \texttt{FieldId} \hookrightarrow Val$	$heap \in Heaps$
$state = \langle stack, heap \rangle$	$state \in States$
$L \triangleq (\mathbb{N} \times \texttt{VarId}) \uplus (Obj \times \texttt{FieldId})$	
$V \triangleq Val$	
$\sigma = heap \uplus \bigcup_{i=1}^{n} \{i\} \times \rho_i$	
Abstract Semantics	
$\sigma^{\#}\colon L^{\#} \hookrightarrow V^{\#}$	$\sigma^{\#} \in \Sigma^{\#}$
$\tau^{\#} \triangleq \langle \sigma^{\#}, p, \sigma'^{\#} \rangle$	$\tau^{\#} \in T^{\#}$

Figure 5. Summary of notations for semantic domains, where Obj is an unbounded set of dynamically allocated objects, `VarId` is the set of local variable identifiers, and `FieldId` is the set of field identifiers

Instruction	Read Set	Write Set
`putfield(b,f,v)`	`v`	$\langle \rho(\texttt{b}), \texttt{f} \rangle$
`r=getfield(b,f)`	$\langle \rho(\texttt{b}), \texttt{f} \rangle$	`r`
`r=checkcast(x,T)`	`x`	`r`
`r=unaryop(¬,v)`	`v`	`r`
`r=binaryop(+,x,y)`	`x, y`	`r`

Figure 6. Read and write sets for several of the Java bytecode instructions on a concrete state $\langle stack, heap \rangle$, where ρ is the top frame on the stack

ADT Operation	Read Set	Write Set
`m.put(k,v):r`	$\varnothing$	$\{ \langle m, k \rangle, \texttt{r} \}$
`m.get(k):r`	$\{ \langle m, k \rangle \}$	$\{ \texttt{r} \}$
`m.containsKey(k):r`	$\{ \langle m, k \rangle \}$	$\{ \texttt{r} \}$
`m.remove(k):r`	$\varnothing$	$\{ \langle m, k \rangle, \texttt{r} \}$

Figure 7. Conservative abstract read and write sets of several operations of the `Map` ADT on abstract state $\sigma^{\#}$, where $m = \rho(\texttt{m})$, $k = \rho(\texttt{k})$ and $v = \rho(\texttt{v})$ (ρ being the top stack frame)

5.1 Concrete Dependencies

We assume a standard concrete semantics for sequential programs, where a state consists of a stack of method invocations mapping local variables to values, and a heap mapping object fields to values (see Fig. 5).

A memory location is a data storage that can be used by the instructions of the programming language to store and retrieve data values. In our semantics, there are two types of memory locations: *heap* and *environment* locations. A heap location is an object's field. In state $\langle stack, heap \rangle$, each pair $\langle o, f \rangle$ in $dom(heap)$ defines a unique memory location. An environment location is a local variable in the context of an invocation of a method (i.e., a stack frame). In state $\langle stack, heap \rangle$, where $stack$ is $\langle \rho_1, \ldots, \rho_n \rangle$, each pair $\langle i, v \rangle$ defines a unique memory location, where $v \in \texttt{VarId}$ is a local variable identifier of a method executed by stack frame i, i.e., $v \in dom(\rho_i)$ and $1 \leq i \leq n$.

To apply the uniform definition of dependencies from Section 4.1, we encode concrete states of the form $\langle stack, heap \rangle$ as states in the low-level semantics by simply mapping from memory locations to values. Given a state $\langle state, heap \rangle$, we define the low-level mapping σ to be the (disjoint) union of the mapping defined by $heap$ and the mappings defined by ρ_i for $i = 1, \ldots, n$, as shown in Fig. 5. This can be extended in the usual way to handle threads, types, arrays and static variables.

Fig. 6 shows read and write sets for several interesting Java bytecode instructions. These sets are conservative. For example, consider a transition τ for `putfield(b,f,v)` from a state where the value of `b.f` before the update is already $\rho(\texttt{v})$. The best write set for this transition does not contain memory location $\langle \rho(\texttt{b}), \texttt{f} \rangle$, because the value at $\langle \rho(\texttt{b}), \texttt{f} \rangle$ has not changed. The best read set for this transition contains memory location $\langle \rho(\texttt{b}), \texttt{f} \rangle$, because by changing the value of this location, the best write set changes. The read set of this transition according to Fig. 6 does not contain $\langle \rho(\texttt{b}), \texttt{f} \rangle$, but the write set does. Fig. 6 is therefore conservative: The read and write sets it prescribes may give rise to more dependencies than the best read and write sets.

In return, the specification in Fig. 6 is simpler than (3) and (4), because the read set in Fig. 6 depends only on the state in which the statement executes, whereas (4) depends on all possible executions of the same statement. In the above example, the best read set takes into account that there exists another state in which the value of $\langle \rho(\texttt{b}), \texttt{f} \rangle$ is not $\rho(\texttt{v})$. For a transition that starts in that state, $write$ contains $\langle \rho(\texttt{b}), \texttt{f} \rangle$, and therefore differs from the best write set for τ.

5.2 Abstraction of Trace Segments

We define an abstraction of trace t by breaking the sequence of transitions constituting t into multiple segments whose concatenation is the original trace. Each segment is mapped to an abstract transition, and intermediate states between the

concrete transitions represented by the abstract transition are omitted.

Let trace t be $\langle \tau_1, \ldots, \tau_n \rangle$. We define segments of t using a set $I = \{a_1, b_1, \ldots, a_k, b_k\}$ of indexes, where

$$1 = a_1 \leq b_1, b_1 + 1 = a_2 \leq b_2, \ldots a_k \leq b_k = n$$

For $i = 1, \ldots, k$, the i-th segment of t consists of transitions between indexes a_i and b_i, inclusive. The abstraction of t is the trace $I(t) \triangleq \langle \overline{\tau}_1, \ldots, \overline{\tau}_k \rangle$, such that $\overline{\tau}_i \triangleq \langle \sigma_{a_i}, \overline{p}_i, \sigma'_{b_i} \rangle$ and $\overline{p}_i$ is a sequential composition of the statements in the i-th segment of t.

A read (*resp.* write) set of an abstract transition $\overline{\tau}_i$ can be naturally defined as the union of all the read (*resp.* write) sets of the concrete transitions in segment i of t:

$$\begin{aligned} \overline{w}(\overline{\tau}_i) &= \bigcup_{a_i \leq j \leq b_i} w(\tau_j) \\ \overline{r}(\overline{\tau}_i) &= \bigcup_{a_i \leq j \leq b_i} r(\tau_j) \end{aligned} \tag{9}$$

This definition is conservative with respect to the best read and write sets: $read(\overline{\tau}_i) \subseteq \overline{w}(\overline{\tau}_i) \cup \overline{r}(\overline{\tau}_i)$ and $write(\overline{\tau}_i) \subseteq \overline{w}(\overline{\tau}_i)$ for $i = 1, \ldots, k$ assuming that w, r are conservative. For the definition of the best read set of $\overline{\tau}_i$ according to (4), we use the semantics of sequential composition to compute the effect of $\overline{p}_i$ on $\widehat{\sigma}$.

Next, we define two useful instances of trace segments abstraction by specifying different ways of partitioning a trace into segments.

Loop-based Segments Segments can be defined by loop iterations. We use this for loop parallelization, because we are interested in dependencies between different loop iterations only, and not between statements within the iteration body.

Method-based Segments Segments can be defined by calls and returns to certain methods, e.g. methods implementing ADT operations. Instructions executed by the body of such a method, including instructions executed by methods called indirectly from it, are represented by a single transition in the abstract trace. Dependencies computed from the result of this abstraction are between ADT operations, but based on concrete memory locations, which reflect the internals of an ADT implementation. To abstract away from internals, we combine this abstraction with an abstraction of states, described in the next section.

5.3 Abstraction of States

The definition of dependencies from Section 4.1 can be applied to any abstract trace that satisfies a simple requirement whereby abstract states essentially map (abstract) memory locations to (abstract) values: $\sigma^{\#} \colon L^{\#} \hookrightarrow V^{\#}$. Intuitively, abstract memory locations enable more precise approximation of commutativity between statements. Instead of testing for conflicing accesses to concrete memory, we consider a more abstract notion of conflict that stems from a semantic dependency between the statements.

Let $\beta \colon \Sigma \to \Sigma^{\#}$ be an abstraction function mapping concrete to abstract states (see examples of β in Section 6).

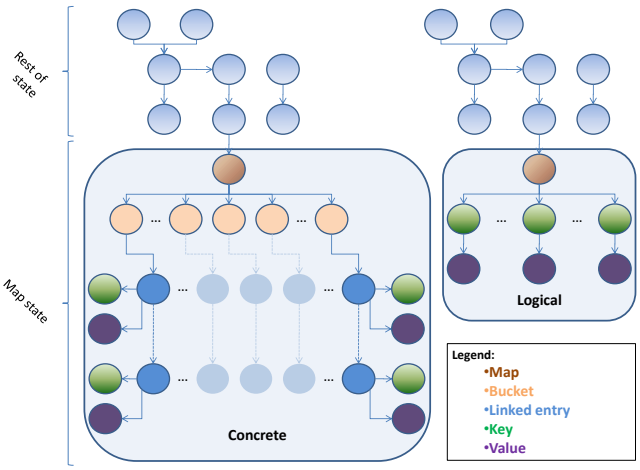

Figure 8. Concrete versus logical representation of a `Map` instance

Let $t = \langle \tau_1, \ldots, \tau_n \rangle$ be a trace. The abstraction of t using β is the trace $t^{\#}_{\beta} \triangleq \langle \tau^{\#}_1, \ldots, \tau^{\#}_n \rangle$, where for $i = 1, \ldots, n$, $\tau^{\#}_i \triangleq \langle \beta(\sigma_i), p, \beta(\sigma'_i) \rangle$. (In what follows, we omit the β subscript when β is clear from the context.)

A transition $\tau^{\#} = \langle \sigma^{\#}, p, \sigma'^{\#} \rangle$ is in $T^{\#}$ if and only if there exist $\sigma, \sigma' \in \Sigma$ such that $\beta(\sigma) = \sigma^{\#}$, $\beta(\sigma') = \sigma'^{\#}$ and $\sigma' = [\![p]\!](\sigma)$.

Note that we cannot compare read (or write) sets of concrete and abstract traces, because these sets refer to different domains of memory locations, L and $L^{\#}$ (unless abstraction α has some special properties, which we do not require here). However, dependencies are comparable, because there is a one-to-one correspondence between the transitions in a given trace and its abstraction (when we ignore cm annotations of memory locations on dependencies).

6. Representation Functions

The abstractions presented in Sections 5.2–5.3 provide the basis for defining and utilizing ADTs, which is the focus of this section. To this end, we introduce a simple notion of representation allowing the programmer to define a logical view for a concrete store [18], as illustrated in Fig. 8.

A key question in the representation of data structures is what type of encapsulation to enforce on them to guarantee representation independence. Another question is how to express representation functions (or relations). In this paper, we employ a restrictive definition of a representation function based on the notion of ownership as domination [29]. Considering concrete trace t and ADT a with set ops of operations, we assume our ability to interpret invocation statements in (transitions in) t as operations from ops. We further assume that the concrete implementation of an ADT is rooted at some base object. The internal state of the ADT implementation is only accessible via the root object. For example, in call `m.put(k,v)` corresponding to the `put` op-

eration of the `Map` ADT, argument `m` points to the root of the `Map` implementation.

A representation function for ADT a, rep_a, operates on a concrete state σ by replacing certain concrete memory locations by a set of abstract locations. Intuitively, the removed locations comprise the concrete state of data structures implementing a, and the abstract locations introduced in their stead represent the abstract state of these data structures according to a. rep_a operates only on heap locations. rep_a may operate on heap location $m = \langle o, \mathtt{f} \rangle$ defined in σ only if it is owned by the root r of some implementation of a, where ownership is equated with domination: Every heap path in σ (consisting of zero or more edges) reaching o goes through r. For simplicity, we assume that ADT operations do not access heap locations outside the ADT.

We note that more permissive definitions of a representation function can be used at the cost of complicating the formal discussion [10]. For our needs, the above definition suffices. It is also compatible with the ADTs we defined for our evaluation (described in Section 7), at least with regard to the traces we considered.

6.1 Abstraction via Representation Functions

Based on the specifications of ADTs $a_1, \ldots, a_k$, we would like to define a state abstraction function $\beta \colon \Sigma \to \Sigma^{\#}$. For this, we assume that for all $i \neq j$, a_i and a_j operate on distinct memory locations and produce disjoint sets of abstract memory locations. We can thus define β as follows:

$$\beta = \lambda\sigma \in \Sigma.(\sigma \setminus \biguplus_{1 \leq i \leq k} rem(rep_{a_i}, \sigma)) \cup \biguplus_{1 \leq i \leq k} add(rep_{a_i}, \sigma)), \tag{10}$$

where $rem(rep_a, \sigma)$ denotes the set of concrete locations defined by σ that are abstracted away by rep_a, and $add(rep_a, \sigma)$ the set of abstract locations introduced by rep_a.

Parallelization with ADTs Information collected about dependencies between ADT operations is actionable for parallelization assuming that the following two properties hold:

1. **Encapsulation.** If the client treats the ADT as such, and is unaware of its implementation details, then the existing ADT implementation can be replaced with relative ease. This amounts to the requirement that the memory locations the representation function operates on are not manipulated outside of ADT operations. (That is, these locations are neither read nor written by transitions not corresponding to operations of the ADT.) This can easily be verified for a single trace (or finitely many traces).
2. **Atomicity.** As discussed in Section 3, we assume an interleaving semantics of concurrency. The trace $I(t)$ after method-based segments abstraction according to ADT operations represents (long) sequences of concrete instructions as single ADT invocations. Reasoning about interleavings at the level of ADT invocations (i.e., entire segments), and not their constituent instructions, assumes that the invoked operations execute without interruption. For a transformation based on this to be valid, the ADT operations must be atomic. This guarantee can be provided by a linearizable [17] implementation of the ADT.

Given that these two requirements are met, Lemma 4.2 holds for $I(t)$.

Basic Computation of Abstract Read and Write Sets

Inputs:
  $\{\langle a_1, rep_{a_1}\rangle, \ldots, \langle a_k, rep_{a_k}\rangle\}$: representation functions
  $\overline{\tau} = \langle \sigma, p, \sigma' \rangle$: transition in $I[a_1, \ldots, a_k](t)$
  $[r, w]$: read and write sets

ABSREADWRITE:
  $\overline{\tau}^{\#} = \text{ABS}(\overline{\tau})$
  return READWRITE($\overline{\tau}^{\#}$)

ABS($\overline{\tau}$):
  for each $a_i \in \{a_1, \ldots, a_k\}$
    if $\overline{\tau} \in \overline{T}_{a_i}$ return $\langle \beta[rep_{a_i}](\sigma), p, \beta[rep_{a_i}](\sigma') \rangle$
  return $\overline{\tau}$

READWRITE($\overline{\tau}^{\#}$):
  if $\overline{\tau} \in \overline{T}$ return $\langle args(\overline{\tau}^{\#}) \cup abs(\overline{\tau}^{\#}), write(\overline{\tau}^{\#}) \rangle$
  else return $\langle r(\overline{\tau}^{\#}), w(\overline{\tau}^{\#}) \rangle$

Figure 9. Basic algorithm for computing abstract read and write sets using representation functions

6.2 Computing Dependencies

To compute dependencies in a sound manner relative to (4) and (3), we must introduce a tractable approximation of (4), which is not computable in general (e.g., if $\Sigma^{\#}$ is infinite). Assuming encapsulation, as well as disjointness between the operations of distinct ADTs, we can use the algorithm in Fig. 9 for computing the abstract read and write sets of a transition, where (i) $\overline{T}_a$ is the set of (segment) transitions corresponding to ADT a and $\overline{T} = \bigcup_{i=1}^{k} \overline{T}_{a_i}$, (ii) $\beta[rep_a]$ denotes the state abstraction function obtained according to (10) by using (only) rep_a, (iii) $args$ is an auxiliary function returning the set of all environment locations used and defined by the statement in its argument transition, and (iv) abs returns the set of abstract locations defined by the entry (abstract) state of its argument transition.

Importantly, the algorithm in Fig. 9 uses rep_a, and not $rep_{a_1}, \ldots, rep_{a_k}$, for abstracting the states of a transition corresponding to an operation of ADT a, and does not apply abstraction at all to compute the read and write sets of concrete transitions. The following claim asserts that no dependencies are lost.

Lemma 6.1. *Let $Rep = \{rep_{a_1}, \ldots, rep_{a_k}\}$ be representation functions for ADTs $a_1, \ldots, a_k$, and r, w read and write sets that agree with* ABSREADWRITE*, instantiated with Rep, on the transitions in segments abstraction $I[a_1, \ldots, a_k](t)$ of concrete trace t. Then if we assume that*

(i) all ADTs are encapsulated, and (ii) there is no sharing between ADTs in terms of their operations, then

$$D[read, write](I[a_1, \ldots, a_k](t)^{\#}) \subseteq D[r, w](I[a_1, \ldots, a_k](t)),$$

where $I[a_1, \ldots, a_k](t)^{\#}$ *is obtained by applying* $\beta[a_1, \ldots, a_k]$ *to* $I[a_1, \ldots, a_k](t)$.

Proof. See Appendix C. □

A primary source of imprecision in ABSREADWRITE is its conservative resolution of the read set of an ADT operation as the entire state of the ADT at the entry state, which consists of all the abstract locations introduced by the ADT's reprsentation function. Another concern, related to performance, is that computing the write set can be expensive, as it entails exhaustive comparison between the abstract states before and after an operation. We now address these concerns.

6.2.1 Exploiting the Concrete Frame

Consider abstract transition $\overline{\tau} = \langle \sigma, p, \sigma' \rangle \in I[a_1, \ldots, a_k](t)$ corresponding to ADT a_i and its respective read and write sets according to (9), $r(\overline{\tau})$ and $w(\overline{\tau})$. We rewrite the entry and exit states of $\overline{\tau}$ as follows:

$$\widetilde{\sigma} = \sigma \setminus \{\langle l, \sigma(l) \rangle \mid l \in dom(\sigma) \setminus (r(\overline{\tau}) \cup w(\overline{\tau}))\}$$
$$\widetilde{\sigma'} = \sigma' \setminus \{\langle l, \sigma'(l) \rangle \mid l \in dom(\sigma') \setminus (r(\overline{\tau}) \cup w(\overline{\tau}))\}$$

That is, we omit the frame of $\overline{\tau}$ from the entry and exit states, which yields entry and exit states $\widetilde{\sigma}$ and $\widetilde{\sigma'}$ that are by definition indistinguishable from σ and σ', respectively, from the perspective of τ.

Applying rep_{a_i} to $\widetilde{\overline{\tau}} = \langle \widetilde{\sigma}, p, \widetilde{\sigma'} \rangle$, and not to $\overline{\tau}$, is preferable since portions of the concrete implementation(s) of a are absent from $\widetilde{\sigma}$ and $\widetilde{\sigma'}$. This suggests a smaller abstract read set, as well as more efficient computation of the abstract write set.

While reliance on concrete memory accesses may improve the accuracy of the analysis, it comes at the cost of tracking concrete reads and writes during the execution of ADT operations. Moreover, there is still the threat that the implementation of an ADT operation is naïve in that its footprint is much larger than the best read and write sets.

Consider for example an implementation of the `Map` ADT where the `get` operation performs linear traversal of all the keys stored in the map until a match is found. In the worst case, the concrete read set of the `get` implementation may translate into an abstract read set consisting of the entire state of the `Map`, which is not much of an improvement compared to the read set prescribed by ABSREADWRITE. This can be remedied by capturing the frame of the `get` operation at a more abstract level.

6.2.2 Exploiting the Abstract Frame

The semantics of ADT operation op may enable partitioning the abstract state immediately preceding its invocation into two disjoint areas corresponding to (a safe approximation of) the frame and the footprint of op, where the abstract locations in op's frame are not needed to establish a sound commutativity specification. This is exemplified by the partial specification in Fig. 7 provided for the `Map` ADT.

Importantly, the `Map` spcification in Fig. 7 relies on a prohibitive representation of the state of a `Map` where a `Map` instance m contains a field k for every allocated object k, and not only for stored keys. This allows, e.g., a sound yet precise read set for `containsKey` even if the queried key is not stored in the concrete `Map` instance. A sound read set can also be obtained using a more concise description of a `Map`, but it would be less precise than the one in Fig. 7 in terms of its corresponding commutativity specification.

Knowledge of ADT semantics enables specialization of the representation function by taking into account the performed operation along with its arguments. In practice, this translates into specialization of the representation function based on the call statements in the trace in the context of their entry state.

Since at each point we need only represent the abstract footprint of an operation, we can implicitly define an abstract state whose explicit representation is prohibitive by referring only to parts of it when describing the footprint of an operation, as is done in Fig. 7. Even when explicit representation of the abstract state is possible, there is still an important gain by considering only a (small) portion of it for the reasons mentioned in Section 6.2.1; namely, a smaller read set, as well as more efficient computation of the write set.

6.2.3 Advanced Version of ABSREADWRITE

An overload of ABSREADWRITE incorporating the enhancements described above is presented in Fig. 10. Note that conceptually, CONCFILTER and SPECIALIZE commute. We can either start by removing the concrete frame of the transition, and then build an abstraction of the entry and exit states that also takes into account the abstract frame of the transition (as shown in Fig. 10), or vice versa. This is because the concrete implementation is known to comply with the ADT's semantics, and so the concrete footprint already enables a safe approximation of the abstract locations needed toward sound commutativity constraints.

Further note that both CONCFILTER and SPECIALIZE are optional, and moreover, any combination of these enhancements is valid. By letting $r(\overline{\tau}) = L$ and $w(\overline{\tau}) = L'$, we effectively disable the concrete-frame heuristic. Analogously, by letting $\nu(\overline{\tau}) = rep_a$ where $\overline{\tau}$ corresponds to an operation of ADT a and rep_a is the generic representation function of a, we skip the abstract-frame representation.

Enhanced Computation of Abstract Read and Write Sets

Inputs:
$\overline{\tau} = \langle \sigma, p, \sigma' \rangle$: transition in $I[a_1, \ldots, a_k](t)$
$\nu = [\overline{\tau} \rightarrow rep_{\overline{\tau}} \mid \overline{\tau} \in \overline{T}]$: specialized representations
$[r, w]$: read and write sets

ABSREADWRITE:
if $\overline{\tau} \notin dom(\nu)$ return $\langle r(\overline{\tau}), w(\overline{\tau}) \rangle$
$\widetilde{\overline{\tau}} = \text{CONCFILTER}(\overline{\tau})$
$\widetilde{\overline{\tau}}^{\#} = \text{SPECIALIZE}(\widetilde{\overline{\tau}})$
return $\langle args(\widetilde{\overline{\tau}}^{\#}) \cup abs(\widetilde{\overline{\tau}}^{\#}), write(\widetilde{\overline{\tau}}^{\#}) \rangle$

CONCFILTER($\overline{\tau}$):
$\widetilde{\sigma} = \sigma \setminus \{\langle l, \sigma(l) \rangle \mid l \in dom(\sigma) \setminus (r(\overline{\tau}) \cup w(\overline{\tau}))\}$
$\widetilde{\sigma'} = \sigma' \setminus \{\langle l, \sigma'(l) \rangle \mid l \in dom(\sigma') \setminus (r(\overline{\tau}) \cup w(\overline{\tau}))\}$
return $\langle \widetilde{\sigma}, p, \widetilde{\sigma'} \rangle$

SPECIALIZE($\overline{\tau}$):
return $\langle \beta[\nu(\overline{\tau})](\sigma), p, \beta[\nu(\overline{\tau})](\sigma') \rangle$

Figure 10. Advanced version of the algorithm for computing abstract read and write sets using representation functions

Computation of Loop-carried Dependencies

Inputs:
t: concrete trace
$\{a_1, \ldots, a_k\}$: ADT specifications
l: loop identifier
$[r, w]$: read and write sets

DEPENDENCIES:
$\langle t_m, t_l \rangle = \text{SEGMENTS}(t, \{a_1, \ldots, a_k\}, l)$
$G(t_m) = \text{DEPGRAPH}(M(t), [r, w])$
$G(t_l) = \text{DEPGRAPH}(L(M(t)), [r, w])$
return
$\{\langle \tau_1, \tau_2 \rangle \in E_{G(t_m)} \mid \langle \tau_1', \tau_2' \rangle \in E_{G(t_l)}.\tau_1 \in \tau_1' \wedge \tau_2 \in \tau_2'\}$

SEGMENTS($t, \{a_1, \ldots, a_k\}, l$):
$M(t) = \text{METHODSEGMENTS}(t, \{a_1, \ldots, a_k\})$
$L(M(t)) = \text{LOOPSEGMENTS}(M(t), l)$
return $\langle M(t), L(M(t)) \rangle$

DEPGRAPH($t, [r, w]$):
$V = \{\tau \in t\}$
$E = \{\langle \tau_1, \tau_2 \rangle \in D[r, w](t)\}$
return $\langle V, E \rangle$

Figure 11. Algorithm for computing loop-carried dependencies

7. Implementation and Evaluation

In this section, we first provide details about the implementation of HAWKEYE (Section 7.1). We then describe two sets of experiments we conducted to evaluate HAWKEYE's utility. In the first experiment (Section 7.3), we measure the reduction in reported dependencies due to semantic conflict detection. The second experiment (Section 7.4) goes beyond these raw numbers to assess the value of HAWKEYE as an aid in end-to-end parallelization.

7.1 Implementation Details

HAWKEYE is implemented atop Chord [1], an extensible static and dynamic program-analysis framework for Java bytecode based on the Joeq compiler infrastructure [37] and the Javassist bytecode-instrumentation library [12]. HAWKEYE accepts as input (i) a program description in the form of a Chord properties file, which includes the program's class files, main entry point, and input data for one or more runs of the program, (ii) a (dynamic) parallelization scope where dependencies should be tracked (e.g., a loop, a method-stack configuration, etc), and (iii) a specification for one or more ADTs.

The algorithm implemented by HAWKEYE is illustrated in Fig. 11. For clarity, the algorithm in Fig. 11 is a slightly simplified version of the actual HAWKEYE algorithm, where the parallelization scope is a single loop executing at most once. This suffices for the experiments described in this section. The dependencies computed by the analysis are at the granularity of ADT operations (and not loop iterations): Each of the transitions involved in a dependency is either an ADT operation or a concrete transition. The loop-based segments abstraction is used to eliminate dependencies where both transitions map to the same loop iteration.

7.2 Experimental Setup

We performed our experiments with the IBM J9 V1.6.0 VM running on 32-bit Linux atop a Lenovo X201 laptop with 4GB of RAM. ADTs from the Java Collections Framework, as well as the graph ADTs in JGraphT and Boruvka, were specified using the abstract-frame enhancement. We manually specified the abstract footprint of operations exposed by these ADTs, as depicted in Fig. 7. For all the remaining ADTs (and benchmarks), a single representation function was defined, and the analysis inferred a refined read set according to the concrete-frame heuristic described in Section 6.2.1.

7.3 Number of Reported Dependencies

The purpose of the first experiment was to gain a quantitative measure of the decrease in the number of reported dependencies under abstraction. Our research hypotheses were the following:

1. **Importance of abstraction.** There is a significant gap between the number of dependencies reported with and without abstraction.
2. **Important of user ADTs.** Use of user-provided representation functions is significant in dependency reduction compared to the baseline analysis that relies solely on built-in representations for library ADTs.

Name	Ver.	Description	Trace Len.
Boruvka	2.1	Solver for the minimum spanning tree problem	813,382
PMD	4.2	Java source code analyzer	2,190,213
JGraphT [4]	0.8.1	Graph library	710,580
JFileSync [3]	2.2	Utility for synchronizing pairs of directories	1,733,552
Weka [6]	3.6.4	Machine-learning library for data-mining tasks	17,945,255
Cobertura [2]	1.9 (rc2)	Java code coverage analysis	2,629,457
WebLech [5]	0.0.3	Web-site download and mirror tool	4,840,544

Figure 12. Benchmark characteristics

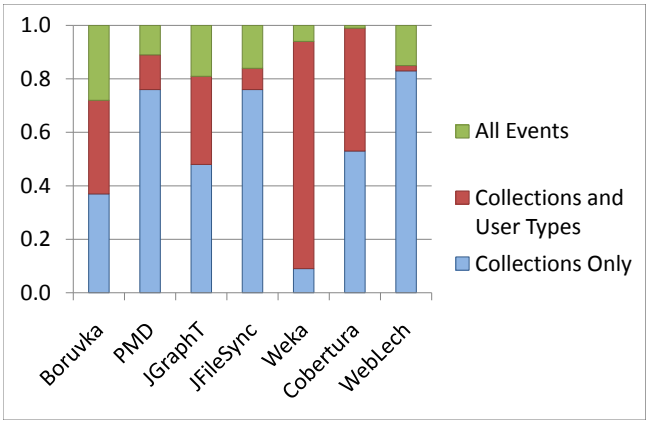

Figure 13. Visualization of the proportion of ADT operations in the subject loops

To test these hypotheses, we implemented a variant of HAWKEYE, which we dubbed MOLEYE, that differs from HAWKEYE only in the value it assigns to $[r, w]$: Where HAWKEYE uses abstract read and write sets according to ADT semantics, MOLEYE relies on (9), which amounts to "localizing" concrete dependencies within ADT calls to their corresponding call sites.

We then ran both analyses on seven real-world benchmarks. The benchmarks' details appear in Fig. 12. From each benchmark, we selected a single loop as the analysis' parallelization target, as follows: We first defined the benchmark's entry point (either its main method or a unit test exercising its core functionality). We then ran a profiling analysis several times, each time with a different input. Loops whose number of iterations was not a function of the input (e.g., initialization loops) were discarded. Of the surviving loops, we chose the one with the highest average number of instructions per iteration.

The analyses were ran in two configurations:

1. The "Collections Only" (CO) configuration treats only Java collections (such as `Set`, `Map` and `List`) as ADTs.
2. The "Collections and User Types" (CU) configuration subsumes the CO configuration by also treating certain interfaces and classes in user code as ADTs. These ADTs are benchmark specific.

Results Table 1 summarizes the results. For each configuration, we provide the results by both analyses according to two counts: The "All Dep.s" count is the total number of dependencies reported by the analysis according to the algorithm in Fig. 11. "ADT Dep.s" considers the subset of all dependencies where both the source and the target transitions occurred during ADT operations. Dependencies outside ADT operations, such as those involving the induction variable, are not included in the "ADT Dep.s" count. Fig. 13 visualizes the differences between the CO (blue) and CU (blue and red) configurations in trace coverage.

7.4 End-to-end Parallelization using HAWKEYE

The second experiment was designed to provide qualitative insight into the value of HAWKEYE as a user aid in end-to-end parallelization. For this, we selected three of the seven benchmarks used for the first experiment.

We then applied the following parallelization methodology:

1. Run HAWKEYE on several representative execution traces.
2. Perform shallow review of the reported dependencies to decide whether additional (user) ADTs should be specified. If further ADTs are required, then augment the specification and return to the first step.
3. Perform in-depth review of the surviving dependencies, and — if needed — transform the code to address these dependencies. If the program was changed, then return to the first step using execution traces of the transformed program.
4. Manually verify the correctness of the transformed program.

PMD PMD is part of the DaCapo suite [9]. It accepts a set of Java classes as input, and analyzes them for a range of source-code problems. PMD has both a sequential and a parallel version. In the concurrent version of PMD, different classes are analyzed in parallel to each other.

Our first run of HAWKEYE on PMD yielded close to 300 dependencies. Many of these dependencies involved the `Report` and `BenchmarkResult` classes, which motivated treatment of these types as ADTs. Rerunning the analysis with the augmented specification yielded many fewer depedencies, of which five were due to operations of `BenchmarkResult` and `Report`. These five dependencies turned out to be important (e.g., conflicting updates to global counters maintained by `BenchmarkResult`, such as the total time spent in each rule and the number of AST nodes it visited).

We addressed these dependencies by privatizing shared data and merging between its different copies at the end of

Benchmark	Collections Only				Collections and User Types			
	HAWKEYE		MOLEYE		HAWKEYE		MOLEYE	
	All Dep.s	ADT Dep.s	All Dep.s	ADT Dep.s	All Dep.s	ADT Dep.s	All Dep.s	ADT Dep.s
Boruvka	227	90	414	277	51	30	289	268
PMD	289	3	307	21	88	5	103	20
JGraphT	58	3	106	51	40	0	66	26
JFileSync	155	5	178	28	19	13	42	36
Weka	11,859	0	11,859	0	21	0	103	82
Cobertura	448	0	496	48	15	0	18	3
WebLech	48	5	63	20	44	6	58	20

Table 1. Benchmark statistics according to the algorithm in Fig. 11

the parallel loop. We then repeated all steps of the parallelization process, and confirmed that the important dependencies are no longer present and no more code changes are required. Comparison to the parallel version of PMD showed that our transformed program was correct (though the developers of PMD chose to use lock-based synchronization, instead of privatization, for some of the shared data).

JFileSync JFileSync is a utility for comparing and synchronizing pairs of directories. The JFileSync code makes heavy use of design patterns, and in particular, the singleton pattern, which forces sharing.

Running HAWKEYE on JFileSync with only collection representations yielded a rather noisy report containing 160 candidate impediments to parallelization. The vast majority of reported dependencies were traceable to classes `JFSProgress` and `JFSComparisonMonitor`. This led us to add a specification for these classes. With the augmented specification in place, many fewer dependencies were reported. Of the 19 remaining findings, 13 were still due to `JFSProgress` and `JFSComparisonMonitor`. All these dependencies were found to be real, and stemmed from use of these classes as singletons.

To resolve the discovered impediments, we applied privatization transformations that exploded the singleton objects into multiple copies and then, after the main loop, again reduced the separate copies into a single object. We verified that the found dependencies were absent from the resulting program by running the parallelization process from scratch on it. We then manually confirmed that the resulting program was correct by (i) exercising it on various test inputs and (ii) carefully reviewing its code.

WebLech WebLech is a website download and mirror tool that emulates standard web-browser behavior. WebLech supports multithreaded execution, but can also be run sequentially. For our experiment, we disabled all forms of synchronization in WebLech, so that it could be treated as a genuinely sequential application.

The results from running HAWKEYE on WebLech motivated consideration of the `Spider` class as an ADT. This yielded negligible reduction in the number of reported depdendencies, but those were relatively few to begin with (< 50). Manual review of the ADT-related dependencies showed them to be real impediments to parallelization due to dependent accesses to internal `Spider` data structures that must occur atomically (e.g., checking whether a URL is scheduled for download via a `contains` call on the set of scheduled URLs, and attempting to add the URL to the download queue only if the answer is negative).

We transformed WebLech by unifying dependent calls into linearizable `Spider` operations, and then reran the parallelization process on the transformed program. This confirmed that the impediments were no longer present and no further code changes were required. Comparison to the original code of WebLech indicated that the program we have arrived at was correct.

7.5 Discussion

The numbers in Table 1 provide strong confirmation our hypothesis on the importance of abstraction: The overall number of ADT-related dependencies reported with abstraction is 106 in the CO configuration and 54 in the CU configuration, compared to 445 and 455, respectively, without abstraction. With only collection ADTs, 77% of the ADT-related dependencies are suppressed, and if user types are also treated as ADTs, then 89% of the original dependencies are omitted leaving the developer with an average of 7.7 dependencies to review (30 being the maximum).

As for the value of user ADTs, naïve interpretation of the results is misleading. User ADTs yield a more abstract trace (with fewer abstract transitions that represent more concrete transitions) compared to using collection ADTs alone, and so the number of dependencies is bound to decrease regardless of whether user ADTs actually improve matters. Still, the results on JGraphT, Weka and Cobertura are an encouraging indication that this extra specification effort is worthwhile. In the first case, the addition of user ADTs annihilated all ADT-related depedencies, and in the second and third cases, the overall number of dependencies decreased by a considerable factor (of 582 in Weka and 30 in Cobertura) without introducing a single ADT-related dependency. In terms of raw coverage, Fig. 13 indicates that the gap between the CO and CU configurations is nonnegligible with

an average difference of 32% between the two configurations.

Our qualitative study on end-to-end parallelization also supports our hypothesis on user ADTs. Augmenting the specification with user types proved to be key to arriving at a manageable report that can be processed manually, and simultaneously also removing large sources of noise. More generally, we found the methodology used for the second experiment to be highly effective. It required few iterations, allowed us to quickly converge on the real impediments to parallelization, and led us in all three cases to acceptable programs that are amenable to parallelization.

For the transformations we performed — which either reduced sharing via privatization or encapsulated accesses to shared data in ADT operations — we could also verify that the impediments found in the original program were absent from the transformed program. We note, however, that this may not be the case in general. For example, a synchronization transformation that governs unsafe accesses to shared data by locks is likely to preseve dependencies, in which case HAWKEYE may still report (some of) the original dependencies on the transformed program.

8. Related Work

In this section, we survey closely related research on software parallelization. For other studies where abstraction is applied during dynamic analysis, the reader is referred to [24, 25] and references therein. Research on program slicing, where accurate tracking of data dependencies is a key challenge, is discussed in [19, 36] and works they cite.

Parallelizing Compilers Compile-time code parallelization is a long-standing research problem that dates back to the early days of the Fortran compiler, which exploited Fortran's strong aliasing guarantees to prove the safety of its transformations. The SUIF [14] compiler framework is designed to study parallelization in shared-memory and distributed-shared-memory machines, and has been the basis of several parallelization techniques, including *affine partitions* [26] and *linear inequalities* [7].

A survey of compiler optimizations of imperative programs for parallel architectures, which typically rely on tracking the properties of arrays using loop dependence analysis, is provided in [8]. [34] describes how to compute the transitive reduction of the data-dependence relation, an optimization we use in our dynamic analysis.

Commutativity analysis, a parallelization technique that exploits commutativity between operations on objects to uncover parallelization opportunities, is introduced in [30–32]. This technique was implemented in a compilation system as a set of automatic analysis algorithms. Our approach follows a similar motivation, though it bases commutativity judgments on ADT semantics rather than automated analysis of implementation code. Moreover, the instantiation of our approach as a dynamic analysis entails different challenges in performance and accuracy, and consequently also different algorithms.

Transactional Memory Transactional boosting [16, 20] is a methodology developed to avoid redundant conflicts in traditional software transactional memory (STM) systems, which synchornize on the basis of read/write conflicts. Instead of checking for competing memory accesses, transactional boosting promotes the notion of abstract locks: Each invocation of a boosted object (which is assumed to be linearizable) is associated with an abstract lock; two abstract locks conflict if their corresponding invocations do not commute.

The Galois system [23] facilitates parallelization of irregular applications, which manipulate pointer-based data structures like trees and graphs. Galois provides syntactic constructs for expressing optimistic parallelism, as well as a runtime scheme for detecting potentially unsafe accesses to shared memory and performing recovery. Similar to transactional boosting and our approach, Galois bases conflict detection on semantic rather than concrete commutativity between operations, which depends on ADT semantics.

A framework for reasoning about conflict-detection schemes expressed as commutativity conditions is presented in [22]. Different commutativity specifications for the same ADT, which are each phrased as a set of predicates associated with pairs of operations, are mapped into a unified representation, the *commutativity lattice*. The commutativity lattice orders the specifications by the amount of parallelism they permit. [22] also shows ways of systematically constructing commutativity checkers based on commutativity specifications from the lattice.

Our analysis also tracks dependencies at the semantic level (and can thus be used to uncover STM-based parallelization opportunities, as shown for Boruvka), but is not specific to transactional memory. Also, representation functions are a different way of expressing commutativity than the specifications of [22] and the abstract locks of [16].

Profiling The *critical path*, defined as the longest path whose instruction instances must be executed sequentially, is used by [15] as an optimistic measure of available parallelism. Their tool operates on concrete traces, and computes the ratio $\frac{n}{k}$ between trace length (n) and the length of the critical path (k) for several sequential benchmarks from the DaCapo suite [9]. Loops L exhibiting good parallelization potential are reported as promising parallelization candidates, where a loop's potential is defined as the ratio $\frac{\Sigma_{L_i} l(L_i)}{\Sigma_{Li} |L_i|}$, where L_i is an instance of L, and $l(L_i)$ and $|L_i|$ denote the length of L_i's critical path and the overall number of instructions it executed, respectively.

ParaMeter [21] produces *parallelism profiles* for irregular programs iterating over worklists using Galois set iterators [23]. These profiles show how many worklist items can be executed concurrently at each step of the algorithm assuming an idealized execution model. ParaMeter also com-

putes the *parallelism intensity* of an algorithm by dividing the amount of work executed in each round by the total amount of work available for execution at that time, i.e., the size of the worklist.

Determinism The guarantees made by our analysis regarding the parallel execution of a sequential trace relate to an extensive body of research on finding and preventing unwanted behaviors in parallel programs due to nondeterministic thread interleavings. [11] proposes an assertion framework for specifying that regions of a parallel program behave deterministically despite arbitrary thread interleavings. The framework allows a programmer to specify that if block P of parallel code is executed twice (with potentially different schedules), from initial states s_0 and s'_0 satisfying user-provided precondition Pre, then the respective final states s and s' must satisfy user-provided postcondition $Post$.

SingleTrack [35] is a dynamic analysis that verifies a stronger form of determinism, where the parallel computation must be free of both external interference (*external serializability*) and race conditions due to communication between threads (*conflict freedom*). These conditions ensure that every schedule produces bitwise identical results.

The notion of equivalence employed by our analysis (*cf.* Section 3) can be expressed in terms of the Pre and $Post$ predicates of [11], where Pre enforces equality of the entry states and $Post$ demands that the final states modulo the ADT implementations be identical, and the ADT implementations have identical abstract states.

9. Conclusion

Dependence analysis provides useful information for parallelization, but its value is often impaired by conservative reporting of dependencies. We have investigated the benefit of leveraging ADT semantics toward more accurate reporting of dynamic dependencies. Experiments we have conducted on seven real-world benchmarks show that a significant portion of the execution (86%) is spent in ADT operations, and more importantly, the vast majority of concrete dependencies (89%) reported between ADT operations are annihilated when ADT semantics are taken into account, and the remaining dependencies are largely real. Our tool, HAWKEYE, was further demonstrated to be effective as a parallelization aid.

References

[1] Chord: A static and dynamic program-analysis framework for java.

[2] Cobertura: A test coverage tool for java. `//http://cobertura.sourceforge.net`.

[3] The jfilesync java file synchronization utility. `http://jfilesync.sourceforge.net`.

[4] The jgrapht java graph library. `http://www.jgrapht.org`.

[5] The weblech download and mirror tool. `http://sourceforge.net/projects/weblech/`.

[6] The weka machine-learning library. `http://sourceforge.net/projects/weka`.

[7] Saman P Amarasinghe. Parallelizing compiler techniques based on linear inequalities. Technical report, 1997.

[8] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26:345–420, 1994.

[9] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, 2006.

[10] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–223, 2003.

[11] Jacob Burnim and Koushik Sen. Asserting and checking determinism for multithreaded programs. *Commun. ACM*, 53:97–105, 2010.

[12] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 364–376, 2003.

[13] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *toplas*, 3:319–349, 1987.

[14] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29:84–89, 1996.

[15] Clemens Hammacher, Kevin Streit, Sebastian Hack, and Andreas Zeller. Profiling java programs for parallelism. In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, pages 49–55, 2009.

[16] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPOPP*, pages 207–216, 2008.

[17] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, 1990.

[18] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[19] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation*, pages 35–46, June 1988.

[20] Eric Koskinen, Matthew Parkinson, and Maurice Herlihy. Coarse-grained transactions. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 19–30, 2010.

[21] Milind Kulkarni, Martin Burtscher, Rajasekhar Inkulu, Keshav Pingali, and Calin Cascaval. How much parallelism is there in irregular applications? In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 3–14, 2009.

[22] Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. Exploiting the commutativity lattice. In *PLDI*, 2011.

[23] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.

[24] Ondrej Lhoták and Laurie J. Hendren. Context-sensitive points-to analysis: Is it worth it? In *CC*, pages 47–64, 2006.

[25] Percy Liang, Omer Tripp, Mayur Naik, and Mooly Sagiv. A dynamic evaluation of the precision of static heap abstractions. In *OOPSLA*, pages 411–427, 2010.

[26] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Comput.*, 24:445–475, 1998.

[27] Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19:279–285, 1976.

[28] Susan S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1975.

[29] John Potter, James Noble, and David Clarke. The ins and outs of objects. In *In Australian Software Engineering Conference*, pages 80–89, 1998.

[30] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *PLDI*, pages 54–67, 1996.

[31] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A technique for automatically parallelizing pointer-based computations. In *IPPS*, pages 14–22, 1996.

[32] Martin C. Rinard and Pedro C. Diniz. Semantic foundations of commutativity analysis. In *Euro-Par, Vol. I*, pages 414–423, 1996.

[33] Daniel J. Rosenkrantz, Sanjay Goel, S. S. Ravi, and Jagdish Gangolly. Structure-based resilience metrics for service-oriented networks. In *EDCC*, pages 345–362, 2005.

[34] John L. Ross and Mooly Sagiv. Building a bridge between pointer aliases and program dependences. *Nordic Journal of Computing*, 8:221–235, 1998.

[35] Caitlin Sadowski, Stephen N. Freund, and Cormac Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In *ESOP*, pages 394–409, 2009.

[36] Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. Thin slicing. In *PLDI*, pages 112–122, 2007.

[37] John Whaley. Joeq: A virtual machine and compiler infrastructure. *Sci. Comput. Program.*, 57:339–356, 2005.

A. Proof of Lemma 4.1

Let r, w and r', w' be two pairs of read and write sets satisfying that for all $\tau \in T$, $r(\tau) \subseteq r'(\tau) \cup w'(\tau)$ and $w(\tau) \subseteq w'(\tau)$, and consider trace t. Assume that $\langle \tau, \widehat{\tau} \rangle \in D[r, w](t)$. Then by (6) and (7), at least one of the following is true: $w(\tau) \cap w(\widehat{\tau}) \neq \varnothing$, $w(\tau) \cap r(\widehat{\tau}) \neq \varnothing$, or $r(\tau) \cap w(\widehat{\tau}) \neq \varnothing$.

If $w(\tau) \cap w(\widehat{\tau}) \neq \varnothing$, then necessarily $w'(\tau) \cap w'(\widehat{\tau}) \neq \varnothing$. As for the option that $w(\tau) \cap r(\widehat{\tau}) \neq \varnothing$, let $l \in w(\tau) \cap r(\widehat{\tau})$. Then either $l \in w'(\tau) \cap r'(\widehat{\tau})$ or $l \in w'(\tau) \cap w'(\widehat{\tau})$. In both cases, $\langle \tau, \widehat{\tau} \rangle \in D[r', w'](t)$. The third and final option is handled symetrically.

B. Proof of Lemma 4.2

The first part of the claim follows from the observation that given transition $\tau_i = \langle \sigma_i, p_i, \sigma_i' \rangle$, p_{i+1} cannot distinguish between σ_i and σ_i', since all the memory locations it accesses in $\sigma_i' = \sigma_{i+1}$ are also defined in σ_i and have the same value there. The same reasoning applies to p_i.

We prove the second part of the claim by induction on the length n of a shortest sequence of adjacent transpositions required to bring δ (the permutation transforming $\vec{P}(t)$ into $\vec{P}(\widehat{t})$) into order. (Such a sequence is guaranteed to exist.) We further argue that the natural number m satisfying $t \equiv_m \widehat{t}$ is n. For the base case where $n = 1$, we can use the first claim proved above, since the transposed statements are adjacent, and their corresponding transitions (in t) are not dependent. Commutativity between the statements implies that traces t and $\widehat{t}$ have the same final state, and thus $t \equiv_1 \widehat{t}$.

For the induction step, we assume that a shortest sequence of adjacent transpositions transforming δ into the identity permutation is of length $n + 1$. We rely on the fact that we can choose a shortest sequence where at the k-th step, transposition $\pi_k = \langle i, i+1 \rangle$ operates on a descent of δ as modified so far (i.e., $\delta_k(i) > \delta_k(i+1)$, where $\delta_1 = \delta$ and $\delta_{m+1} = \pi_m(\delta_m)$). Let S be such a sequence and $\pi_1 = \langle i, i+1 \rangle$ the first transposition in S. Applying π_1 to $\vec{P}(\widehat{t})$ removes descent i. Observe that for all $\langle \tau_i, \tau_j \rangle \in D[r, w](t)$, $\delta\pi_1(i) < \delta\pi_1(j)$, since the application of π_1 to δ merely removes an inversion. Thus, by the induction hypothesis, if we consider the trace $\widetilde{t}$ corresponding to schedule $\vec{P}(\widehat{t})\pi_1$ run from starting state σ_1, then $t \equiv_n \widetilde{t}$. Since $\widehat{t} \equiv_1 \widetilde{t}$, we conclude that $t \equiv_{n+1} \widehat{t}$, which completes our proof.

C. Proof of Lemma 6.1

We show that if memory location l is in the read set of $\overline{\tau}^{\#} = \langle \sigma^{\#}, p, \sigma'^{\#} \rangle$, then it is either in the read set or the write set computed by AbsReadWrite for $\tau = \langle \sigma, p, \sigma' \rangle$, and if l is in the write set of $\overline{\tau}^{\#}$, then it is in the write set computed by AbsReadWrite for τ. We split our proof into four cases:

Case 1: *l is an environment location, and τ is a non-ADT transition.* Since AbsReadWrite does not apply abstraction and uses conservative read and write sets, l is treated conservatively.

Case 2: *l is an environment location, and τ is an ADT transition.* l is preserved by the relevant abstraction function,

$\beta[rep_{a_i}]$, since rep_{a_i} only operates on heap locations. If l is read by $\overline{\tau}^{\#}$, then it must be an argument of p (or the result, if the value of the defined variable remains unchanged), in which case it is also in the read set assigned to $\overline{\tau}$. If l is written by $\overline{\tau}^{\#}$, then this is because its value has changed between the entry and exit states. AbsReadWrite also uses $write$, and thus l is guaranteed to also be in the write set of $\overline{\tau}$.

Case 3: *l* ***is a heap location, and*** *τ* ***is a non-ADT transition.*** By our assumption of encapsulation, l is a concrete heap location. AbsReadWrite treats l conservatively in refraining from applying abstraction and using conservative read and write sets.

Case 4: *l* ***is a heap location, and*** *τ* ***is an ADT transition.*** By our (simplifying) assumption that ADT operations do not manipulate heap locations outside the ADT, we conclude that l is part of the state of some ADT a. By our assumptions regarding encapsulation and disjointness of ADT operations, we know that it suffices to base β only on rep_a to observe accesses to l in transition $\overline{\tau}^{\#}$. If l is read by $\overline{\tau}^{\#}$, then it is also read under AbsReadWrite since the entire state of a is considered read. If l is written by $\overline{\tau}^{\#}$, then AbsReadWrite also marks it as read since it applies $write$ to $\beta[rep_a](\sigma)$ and $\beta[rep_a](\sigma')$.