A Dynamic Evaluation of the Precision of Static Heap Abstractions

Percy Liang
UC Berkeley
pliang@cs.berkeley.edu

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

Mayur Naik
Intel Labs Berkeley
mayur.naik@intel.com

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

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Measurement, Experimentation, Verification

Keywords heap abstractions, static analysis, dynamic analysis, concurrency

Abstract

The quality of a static analysis of heap-manipulating programs is largely determined by its *heap abstraction*. Object allocation sites are a commonly-used abstraction, but are too coarse for some clients. The goal of this paper is to investigate how various refinements of allocation sites can improve precision. In particular, we consider abstractions that use call stack, object recency, and heap connectivity information. We measure the precision of these abstractions dynamically for four different clients motivated by concurrency and on nine Java programs chosen from the DaCapo benchmark suite. Our dynamic results shed new light on aspects of heap abstractions that matter for precision, which allows us to more effectively navigate the large space of possible heap abstractions.

1. Introduction

Many static analyses of heap-manipulating programs require reasoning about the heap. This reasoning is driven by a *heap abstraction*, a systematic way to partition the typically unbounded number of concrete objects at run-time into a finite set of abstract objects. The choice of heap abstraction impacts the precision and scalability—and ultimately the usability—of a static analysis.

Object allocation sites are perhaps the most popular kind of heap abstraction. Analyses based on this abstraction place all objects allocated at the same site in the program into the same partition. Though useful in some cases, allocation sites are too coarse to prove many properties of interest. Consequently, a plethora of refinements have been proposed in the literature (e.g., [4, 20, 24, 27, 31]). The goal of this paper is to understand which types of refinement are most useful for various clients.

Abstractions We focus on a family of heap abstractions that refine object allocation sites by augmenting the abstraction of an object with information of the following kind:

Call stack We add the chain of the k most recent call sites on the stack of the thread creating the object. The resulting heap abstraction, known as k-CFA [31] with *heap cloning* or *heap specialization*, is popular in points-to analyses for both procedural and object-oriented languages (e.g. C and Java).

Object recency If an object is the i-th to last allocated at its allocation site, the *recency index* of that object is $\min\{i-1, r\}$, where r is a fixed maximum depth. Adding the recency index allows us to distinguish the last r objects created at an allocation site and from all other objects created earlier at that site.[1] This heap abstraction, called the *recency abstraction* [4, 24], is particularly useful for fine-grained reasoning about loops, as it allows distinctions between objects created in different iterations of the same loop.

Heap connectivity We distinguish objects by their connectivity properties in the heap, e.g., by associating an object o with the allocation sites of other objects that can reach o through the heap graph. These *heap connectivity* predicates are common in shape analysis [27], and allow reasoning about complex data structures.

Clients As we will see, the precision of a heap abstraction depends heavily on the client. In this study, we study four clients for Java which are motivated by concurrency—static race and deadlock detection, in particular.

The THREADESCAPE client asks whether a particular heap-accessing statement in the program (that is, an access

[1] Note that in this paper, we use the term *recency* to refer to recency of allocation, not recency of access.

to an instance field or array element) ever accesses objects which are reachable from a static field (and thus potentially accessible by more than one thread). The SHAREDACCESS client asks a stronger question: whether that object is actually accessed by multiple threads. These two clients are useful for static race detection: a statement is race-free if it only accesses thread-local data.

The SHAREDLOCK client is related, asking whether a lock acquisition statement in the program ever holds a lock that is ever held by more than one thread. This client is similarly useful for static deadlock detection: a statement cannot be involved in a deadlock if it holds a lock that is only ever held by one thread.

Finally, the NONSTATIONARYFIELD client asks whether a given instance field f in the program is *stationary* [32], that is, whether for every object o, all writes to $o.f$ precede all reads of $o.f$. This client is again useful for race detection: a pair of read-write statements accessing f cannot race if f is stationary.

Methodology We evaluated our heap abstractions on the clients described above on nine concurrent Java programs from the DaCapo benchmarks suite [5]. While our motivation is ultimately static analysis, our methodology for evaluating our family of abstractions and clients is based on a dynamic analysis. Specifically, a program is run concretely, and abstractions are computed on the fly, against which queries are answered. Working in this setting allows us to focus only on heap abstractions, since we assume that all other aspects static analyses must contend with (e.g., primitive data, destructive updates, merge points, and method summarization) are handled optimally.

This dynamic setting therefore provides an *upper bound* on the precision of the best possible static analysis that uses a given heap abstraction. While we cannot say definitively that an abstraction will work well within a static analysis due to other compounding factors, we can certainly show that a heap abstraction is ineffective for some client. For example, for the THREADESCAPE client on the `luindex` benchmark, only 6.3% of the queries reported to be escaping by using the allocation site abstraction are actually escaping. In this case, the heap abstraction is clearly a bottleneck: no amount of work on the other aspects of static analysis can help.

Summary of Main Results

1. We show that the precision of an abstraction on a client is in part driven by whether the abstraction is in line with the properties of the client. For example, for the NONSTATIONARYFIELD client, RECENCY (defined in Section 4.2) works quite well because both the client and the abstraction involve temporal properties. On the other hand, increasing k has virtually no impact.
2. We evaluated the effect of varying the call site depth k for abstractions based on k-CFA. For THREADESCAPE, we showed that as k increases, the precision undergoes a sharp phase transition, with the critical k value occurring between $k = 3$ and $k = 6$.
3. We found that RECENCY is an important dimension overall, leading to the highest precision on three of the four clients. We also show that increasing the recency depth r improves precision past $r = 1$, which has been the only case studied in past work.
4. We show that adding refinements along multiple dimensions simultaneously can be important. In one case, REACHFROM does not improve precision over ALLOC until we use k-CFA with $k \geq 5$. In another case, increasing the recency depth from $r = 1$ to $r = 2$ is useless unless k is large enough.
5. We use the number of abstract objects (which we call abstraction size) to measure the potential scalability of an abstraction. We found that RECENCY offers the best tradeoff between precision and size.

The rest of the paper is organized as follows. Section 2 formalizes our methodology. Section 3 describes our four clients and Section 4 describes our family of heap abstractions. Sections 5 and 6 describe the benchmarks and experimental setup, respectively. Section 7 presents our results. Section 8 surveys related work, and Section 9 concludes.

2. Methodology

We present the general framework for our empirical study. The basic idea is this: We run a dynamic analysis that maintains the concrete environment, heap, and various other instrumentation. At various points along the execution trace, the abstraction is applied to the concrete state to produce an abstract state, which is used to answer client queries.

2.1 Program Syntax and Semantics

We now present our dynamic analysis. Figure 1 provides the relevant notation. A program is described by a set of program points $\mathbb{P}$, where each program point $p \in \mathbb{P}$ has a statement $stmt(p) \in \mathbb{S}$. Figure 1 provides the full list of statements; For example, an instance field read takes the form $v_1 = v_2.f$ for local variables $v_1, v_2 \in \mathbb{V}$ and instance field $f \in \mathbb{F}$.[2] The statement $spawn\ v$ creates a new thread by calling `java.lang.Thread.start()` with `this` set to the object pointed to by v. The statement $lock\ v$ acquires a lock on the object pointed to by v. We are not concerned with the control-flow graph of the program as our analysis is dynamic and depends only on the execution trace (defined below).

Having established the syntax, we now describe the semantics of program execution. When the program executes, there are a set of *threads* indexed by a set of thread IDs $\mathbb{T}$. For each thread ID $t \in \mathbb{T}$, $\theta(t)$ specifies two pieces of information about that thread: (1) a current program point $\theta(t).p \in \mathbb{P}$ and (2) an *environment* $\theta(t).\rho \in \Lambda$, which maps each local

[2] We use f to denote both instance fields and array indices.

Syntactic domains:

$$
\begin{array}{rrcl}
\text{(program point)} & p & \in & \mathbb{P} \\
\text{(static field)} & g & \in & \mathbb{G} \\
\text{(local variable)} & v & \in & \mathbb{V} \\
\text{(instance field or array index)} & f & \in & \mathbb{F}
\end{array}
$$

$$
\begin{array}{rrcl}
\text{(primitive stmt.)} & s & \in & \mathbb{S} \\
 & s & ::= & v = null \mid v = new \\
 & & \mid & v_1 = v_2 \mid v = g \mid g = v \\
 & & \mid & v_1 = v_2.f \mid v_1.f = v_2 \\
 & & \mid & spawn\ v \mid lock\ v \\
\text{(stmt. at point)} & stmt & \in & \mathbb{P} \to \mathbb{S}
\end{array}
$$

Semantic domains:

$$
\begin{array}{rrcl}
\text{(object)} & o & \in & \mathbb{O} \\
\text{(environment)} & \rho & \in & \Lambda = \mathbb{V} \to (\mathbb{O} \cup \{null\}) \\
\text{(thread ID)} & t & \in & \mathbb{T} \\
\text{(threads)} & \theta & \in & \Theta = \mathbb{T} \to (\mathbb{P} \times \Lambda) \\
\text{(heap)} & \sigma & \in & \Sigma = (\mathbb{O} \times \mathbb{F}) \to (\mathbb{O} \cup \{null\}) \\
\text{(auxiliary instr.)} & u & \in & \mathbb{U} \\
\text{(state)} & \omega & \in & \Omega = \mathbb{T} \times \Theta \times \Sigma \times \mathbb{U} \\
\text{(trace)} & r & ::= & [\omega_1, ..., \omega_n] \\
 & & & \\
\text{(query)} & q & \in & \mathbb{Q} \subset (\Omega \to bool)
\end{array}
$$

Figure 1: Program trace syntax and semantic domains. Let $\omega.t$, $\omega.\theta$, $\omega.\sigma$, and $\omega.u$ correspond to the components of a state ω. For convenience, also define $\omega.p = \omega.\theta(\omega.t).p$ to be the program point of the current thread $\omega.t$, and let $\omega.\rho = \omega.\theta(\omega.t).\rho$ be the environment of thread $\omega.t$.

variable $v \in \mathbb{V}$ to the concrete object $\theta(t).\rho(v) \in \mathbb{O}$ that v points to. All threads share a *heap* $\sigma \in \Sigma$, which is a directed graph whose nodes are concrete objects and edges are pointers labeled with an instance field or array index. In particular, $\sigma(o, f) = o'$ means object $o \in \mathbb{O}$ points to $o' \in \mathbb{O}$ via field $f \in \mathbb{F}$.

At any time during program execution, there is a *state* $\omega \in \Omega$, which contains the following information: (1) the ID $\omega.t \in \mathbb{T}$ of the thread that is about the execute, (2) the threads $\omega.\theta \in \Theta$, (3) the heap $\omega.\sigma \in \Sigma$, and (4) any auxiliary instrumentation $\omega.u \in \mathbb{U}$ which is needed either by the abstraction or client. This instrumentation (detailed in Figure 2) includes information such as object allocation sites and call stack information.

A full program execution is represented by a *trace* r, which is a sequence of *states* $[\omega_1, \ldots, \omega_n]$. We omit the concrete semantics for the various statements as it is standard. The auxiliary instrumentation warrants more discussion, which we defer to Section 3.

For abstractions:

$$
\begin{array}{rrcl}
\text{(allocation site)} & h & \in & \mathbb{H} \\
\text{(allocation site of object)} & as & \in & \mathbb{O} \to \mathbb{H} \\
\text{(objects in order of creation)} & os & \in & \mathbb{O}^* \\
\text{(method call site)} & i & \in & \mathbb{I} \\
\text{(method call stack of object)} & cs & \in & \mathbb{O} \to \mathbb{I}^*
\end{array}
$$

For clients:

$$
\begin{array}{rrcl}
\text{(thread-escaping objects)} & esc & \in & \mathcal{P}(\mathbb{O}) \\
\text{(threads accessing an object)} & accs & \in & \mathbb{O} \to \mathcal{P}(\mathbb{T}) \\
\text{(threads locking an object)} & lcks & \in & \mathbb{O} \to \mathcal{P}(\mathbb{T}) \\
\text{(object-field pairs read)} & rds & \in & \mathcal{P}(\mathbb{O} \times \mathbb{F})
\end{array}
$$

Figure 2: Auxiliary instrumentation $u \in \mathbb{U}$ needed by various abstractions and clients. Let $\omega.esc$, $\omega.accs$, $\omega.lcks$, and $\omega.rds$ denote the instrumentation collected for state ω. Sections 3 and 4 provide the semantics of these quantities.

$$
\begin{array}{rrcl}
\text{(abstract object)} & a & \in & \mathbb{A} \\
\text{(abstraction function)} & \alpha & \in & (\Omega \times \mathbb{O}) \to \mathbb{A} \\
 & & & \\
\text{(abstract env.)} & \rho^\alpha & \in & \Lambda^\alpha = \mathbb{V} \to \mathbb{A} \\
\text{(abstract threads)} & \theta^\alpha & \in & \Theta^\alpha = \mathbb{T} \to (\mathbb{P} \times \Lambda^\alpha) \\
\text{(abstract heap)} & \sigma^\alpha & \in & \Sigma^\alpha = (\mathbb{A} \times \mathbb{F}) \to \mathcal{P}(\mathbb{A}) \\
\text{(abstract aux. instr.)} & u^\alpha & \in & \mathbb{U}^\alpha \\
\text{(abstract state)} & \omega^\alpha & \in & \Omega^\alpha = \mathbb{T} \times \Theta^\alpha \times \Sigma^\alpha \times \mathbb{U}^\alpha \\
\text{(abstract query)} & q^\alpha & \in & \mathbb{Q} \subset (\Omega \to bool)
\end{array}
$$

Figure 3: Abstract versions of our semantic domains.

2.2 Abstractions

An *abstraction function* α, the central object of interest in this paper, maps a concrete object $o \in \mathbb{O}$, in the context of a concrete state $\omega \in \Omega$, to an abstract object $a = \alpha(\omega, o)$ in some abstract domain $\mathbb{A}$ (see Figure 3).[3] Intuitively, α defines an equivalence relation over objects such that objects in the same equivalence class are not distinguished. For example, the classical object allocation site abstraction maps $o \in \mathbb{O}$ to the site $h \in \mathbb{H}$ where o was allocated. However, the abstraction function can in general depend on any aspect of the current state ω, which will be crucial for defining the reachability and recency abstractions (Section 4).

The abstraction function α can be used to map concrete objects to abstract objects, but in order to answer queries, we will use α to map a concrete state ω to an abstract state ω^α. We construct ω^α by applying α to the various parts of $\omega = (t, \theta, \sigma, u)$ as follows:

1. For the current thread ID t, no abstraction is performed.

[3] For convenience, define $\alpha(\omega, null) = null$.

2. For the thread $\theta(t) = (p, \rho)$ with program point p and environment ρ, we define $\theta^\alpha(t) = (p, \rho^\alpha)$, where the abstract environment ρ^α is computed by applying the abstraction α to the object $\rho(v)$ to which a variable $v \in \mathbb{V}$ points:

$$\rho^\alpha(v) = \alpha(\omega, \rho(v)). \tag{1}$$

3. For the heap σ, we define the abstract heap σ^α by collapsing objects in the heap graph that map to the same abstract object:

$$\begin{aligned}\sigma^\alpha(a, f) \quad = \quad & \{a' \in \mathbb{A} : \exists o' \in \mathbb{O}, \sigma(o, f) = o', \\ & a = \alpha(\omega, o), a' = \alpha(\omega, o')\}. \end{aligned} \tag{2}$$

Note that in the abstract graph, there might be more than one edge leaving a node with the same field label.

4. For a given auxiliary instrumentation u, the abstract instrumentation u^α consists of abstracted versions of the information needed for clients (the concrete versions are described in Section 3):

- The abstract escaping set esc^α consists of the set of abstract values taken on by some concrete object in the concrete escaping set esc:

$$esc^\alpha = \{\alpha(\omega, o) : o \in esc\}. \tag{3}$$

- Whereas $accs$ maps a concrete object to the set of threads that have accessed it, $accs^\alpha$ maps an abstract object a to the set of threads that have accessed any concrete object with abstraction a:

$$accs^\alpha(a) = \bigcup_{o:\alpha(\omega,o)=a} accs(o). \tag{4}$$

- Similarly, $lcks^\alpha$ maps an abstract object a to all the threads that have locked any concrete object with abstraction a:

$$lcks^\alpha(a) = \bigcup_{o:\alpha(\omega,o)=a} lcks(o). \tag{5}$$

- Finally, rds^α is the set of all pairs (a, f) such that some object with abstraction a had its field f read:

$$rds^\alpha = \{(\alpha(\omega, o), f) : (o, f) \in rds\}. \tag{6}$$

From these four cases, we can observe the general recipe for constructing abstract instrumentations: for sets whose elements involve objects (e.g., esc and rds), project these objects onto their abstractions; for functions mapping objects to sets (e.g., $accs$ and $lcks$), construct a mapping from abstract values to a union of those sets.

2.3 Answering Queries

A *client* is specified by a set of queries which each operate on a trace. An example of a query for THREADESCAPE is: at program point p, does variable v ever point to an object which is reachable from a static field or was the argument of $spawn$?

It will be useful to formulate a query q on a trace $r = [\omega_1, \ldots, \omega_n]$ in terms of a disjunction over individual queries on each state ω_i in the trace:

$$q(r) = \bigvee_{i=1}^{n} q(\omega_i). \tag{7}$$

For example, the THREADESCAPE query specified by (p, v) is true if at *any* point in the trace where the current statement is p, v points to thread escaping data (see Section 3.1 for more details).

Henceforth, we will consider the client to be defined by a set of queries $\mathbb{Q}$, where each query $q \in \mathbb{Q}$ maps a concrete state ω to a boolean indicating whether a given property holds on that state. These queries induce the quantity of interest via a disjunction over the states in the dynamic trace (7). Note that a static answer to the query would involve a further disjunction over all possible traces of a program.

To evaluate an abstraction with respect to a client, we must be able to answer queries against the abstraction. For a query $q \in \mathbb{Q}$ and an abstraction α, we let $q^\alpha \in \Omega^\alpha \to bool$ denote an *abstract query*.

The optimal abstract query, based on supervaluational semantics, would return true for an abstract state if the concrete query is true for any state ω with that abstract state:

$$q^\alpha_{\text{opt}}(x) = \bigvee_{\omega:\omega^\alpha=x} q(\omega). \tag{8}$$

This query is both sound and complete, but is in general a difficult quantity to compute, so we will present sound approximations—that is, q^α for which the following condition holds:

$$q(\omega) = 1 \quad \Rightarrow \quad q^\alpha(\omega^\alpha) = 1. \tag{9}$$

We measure the quality of an abstraction α by *precision*, the fraction of queries answered true under the abstraction which are actually true concretely:

$$\text{precision}(\alpha, r) = \frac{|\{q \in \mathbb{Q} : q(r)\}|}{|\{q \in \mathbb{Q} : q^\alpha(r^\alpha)\}|}. \tag{10}$$

Note that queries are based on static code artifacts (for example, for THREADESCAPE, queries correspond to all variables at all heap-accessing program points), not on dynamic objects. Therefore, a small change in a single object which is pointed to by many variables can affect many queries, and thus have a large impact on precision. Some of this sensitivity is intrinsic to the clients. For example, in THREADESCAPE, adding a single link in the heap can cause an arbitrary large set of objects to escape.

3. Clients

We study four clients motivated by concurrency. In particular, these clients can be used by higher-level analyses for findings concurrency defects such as races and deadlocks. Some have additional applications which we will discuss later.

We chose the clients to satisfy three requirements: (1) the client should be useful for solving a real-world problem; (2) the client should have a clear evaluation metric based on precision; and (3) the client should require reasoning about the heap and thus depend on the quality of the heap abstraction. For low-level clients with no clear application, it would be hard to appreciate the effect of the abstraction. On the other hand, high-level clients are more problematic from a methodological perspective, as they often require more than a good heap abstraction and might be harder to evaluate.

For each client, we formulate the property of interest in terms of queries of the form $q \in \Omega \to bool$. We show how these queries can be computed form the auxiliary instrumentation. We also define the abstract query q^α.

3.1 THREADESCAPE

A key problem in the analysis of concurrent programs is identifying which data in a program is thread-local, i.e., reachable from at most one thread. Information about thread-locality is useful for reducing false positives in static race and deadlock analyses [25], as well as reducing the runtime overhead of software-transactional memory [35] and dynamic analyses for finding concurrency defects [10]. More efficient memory allocators and garbage collectors for multi-threaded programs, as well as optimizations in multi-threaded programs under sequentially-consistent memory models, can also benefit from thread-locality [15].

One way to obtain this information is by asking *thread-escape* queries: At a program point $p \in \mathbb{P}$, can a given variable $v \in \mathbb{V}$ ever point to an object which is reachable from more than one thread (e.g., by following a series of field pointers from a static field)? This question can be expressed as a disjunction over the states ω in the trace (7) of the following query:

$$\textsc{ThreadEscape}(p, v)(\omega) \triangleq \omega.p = p \wedge \omega.\rho(v) \in \omega.esc. \tag{11}$$

We now define the escaping set esc. The set esc of the initial state ω_1 is empty. Given the set esc of a state ω, the set esc' of the next state ω' is defined to be the set of objects o' which are *reachable* via the heap graph $\omega.\sigma$ (denoted $o \overset{\omega.\sigma}{\rightsquigarrow} o'$) from an object o satisfying any of the following three conditions:

1. $o \in esc$,
2. the current statement sets some static field $g \in \mathbb{G}$ to point to o (that is, $stmt(\omega.p) \equiv g = v$ and $\omega.\rho(v) = o$), or
3. the current statement executed is $spawn\ v$ and v points to o (that is, $stmt(\omega.p) \equiv spawn\ v$ and $\omega.\rho(v) = o$). This condition captures the fact that objects o which are passed into newly created threads (via $spawn$) also escape.

Now, we need to define the set of queries $\mathbb{Q}$. Motivated by race detection, we include query $\textsc{ThreadEscape}(p, v)$ if p is an instance field or array element read/write statement and v is the variable whose field is being accessed, namely $v.f = y$ or $y = v.f$ for some variable y.

Given an abstraction α, the natural abstract query would be as follows:

$$\textsc{ThreadEscape}^\alpha(p, v)(\omega^\alpha) \triangleq \omega^\alpha.p = p \wedge \omega^\alpha.\rho^\alpha(v) \in \omega^\alpha.esc^\alpha. \tag{12}$$

However, this abstract query is costly to evaluate, because we would need to compute graph reachability to ascertain $\alpha(\omega, o) \in \omega^\alpha.esc^\alpha$ for each state ω in the trace. We therefore define a new set $\omega.\overline{esc}$, which is a relaxation of the escaping set $\omega.esc$ (that is, $\omega.esc \subset \omega.\overline{esc}$): The transfer function for $\overline{esc}$ is analogous to that of esc, with the exception that reachability is defined with respect to the abstract heap—that is, o reaches o' iff $\alpha(\omega, o) \overset{\omega^\alpha.\sigma^\alpha}{\rightsquigarrow} \alpha(\omega, o')$.

Intuitively, we are using the abstraction to update the escaping information directly, instead of only using it to answer queries. The resulting query under the relaxation is the same as (12), only with esc^α replaced with

$$\overline{esc}^\alpha \triangleq \{\alpha(\omega, o) : o \in \omega.\overline{esc}\}. \tag{13}$$

Note that $\omega.esc^\alpha \subset \omega.\overline{esc}^\alpha$, so the resulting relaxed abstract query is still sound.

3.2 SHAREDACCESS

Another property that captures the notion of thread non-locality is SHAREDACCESS. Unlike THREADESCAPE, which deems an object to be non-local simply when it is *reachable* from more than one thread, SHAREDACCESS deems an object to be non-local when it is actually *accessed* from more than one thread—a stronger property.

We define an *access* to be an instance field or array element read/write. Intuitively, an object o is considered *thread shared* if there are two states along the execution trace ω_1 and ω_2, such that the two statements $stmt(\omega_1.p)$ and $stmt(\omega_2.p)$ access some field of the same object but are executing under different threads ($\omega_1.t \neq \omega_2.t$). This property is easy to answer given the the auxiliary instrumentation $\omega.accs$, which provides for each object the set of threads that have accessed it:

$$\textsc{SharedAccess}(p, v)(\omega) \triangleq \omega.p = p \wedge |\omega.accs(\omega.\rho(v))| > 1. \tag{14}$$

The set of queries $\mathbb{Q}$ corresponds to all field accessing statements, as in THREADESCAPE (Section 3.1).

We now define the access sets $accs$. For the initial state, $accs(o)$ is empty for each object o. Given the set $accs(o)$ of state ω, the set $accs'(o)$ of the next state ω' includes threads t which satisfy any of the following:

1. $t \in accs(o)$, or
2. t is the current thread ($\omega.t = t$) and a field of o is accessed (that is, $stmt(\omega.p) \in \{x.f = y, y = x.f\}$ for any y and $\omega.\rho(x) = o$).

The abstract query SHAREDACCESS$^\alpha(p, v)$ is answered by seeing if v points to an object with an abstraction $\omega.\rho^\alpha(v)$ that has been accessed by more than one thread:

$$\text{SHAREDACCESS}^\alpha(p, v)(\omega^\alpha) \triangleq \omega^\alpha.p = p \wedge |\omega^\alpha.accs^\alpha(\omega^\alpha.\rho^\alpha(v))| > 1. \quad (15)$$

Recall that $\omega^\alpha.accs^\alpha(a)$ is the union of $\omega.accs(o)$ over objects o with abstraction a.

3.3 SHAREDLOCK

A *thread-shared lock* is an object used by a lock acquisition statement that is executed by more than one thread. The statement $lock\ v$ captures the three cases of lock acquisition in Java: (1) synchronized static methods, in which the lock object is the class object; (2) synchronized instance methods, in which the lock object is the `this` object; and (3) blocks of the form `synchronized(v) { ... }`, in which the lock object is `v`.

One natural application of the SHAREDLOCK client is synchronization removal [2, 3, 6–8, 26]. However, in recent years this problem has been obviated by advances in hardware and JVMs. Static deadlock detection, on the other hand, remains an important problem, which can benefit greatly from knowing which synchronization operations can safely be ignored.

By analogy to SHAREDACCESS, we define SHAREDLOCK by replacing $accs$ with $lcks$:

$$\text{SHAREDLOCK}(p, v)(\omega) \triangleq \omega.p = p \wedge |\omega.lcks(\omega.\rho(v))| > 1. \quad (16)$$

The definition of lock sets $lcks$ is similar to that of $accs$. Initially, $lcks(o)$ is empty. Given the set $lcks(o)$ of state ω, the set $lcks'(o)$ of the next state ω' includes threads t which satisfy any of the following:

1. $t \in lcks(o)$, or
2. t is the current thread ($\omega.t = t$) and a lock is placed on o ($stmt(\omega.p) = lock\ v$ and $\omega.\rho(v) = o$).

The abstract query SHAREDLOCK$^\alpha(p, v)$ is defined according to SHAREDLOCK(p, v), but replacing ω with ω^α, ρ with ρ^α, and $lcks$ with $lcks^\alpha$ in (17).

3.4 NONSTATIONARYFIELD

Stationary fields were first introduced by Unkel and Lam [32] as a generalization of the `final` keyword in Java. A field is considered *stationary* if all instances of the class declaring the field satisfy the property that all writes to the field occur before all reads.

As noted in [32], knowing which fields are stationary provides an object-oriented basis for reasoning about aliasing relations across time; such information can be used, e.g., by a deadlock analysis when reasoning about aliasing between locks stored as object fields. A more immediate application is in race detection, where a pair of read/write statements on the same field f is race-free if f is stationary.

We define the query on the negation of the stationary-field property. A field is non-stationary if there exists a state ω in the trace such that NONSTATIONARYFIELD$(f)(\omega)$ returns true, where NONSTATIONARYFIELD$(f)(\omega)$ returns true if the current statement ($stmt(\omega.p)$) writes to field f of an object o which has been previously read ($(o, f) \in \omega.rds$). Formally:

$$\text{NONSTATIONARYFIELD}(f)(\omega) \triangleq (stmt(\omega.p) \equiv x.f = y) \wedge (\omega.\rho(x), f) \in \omega.rds. \quad (17)$$

We need to define rds. Given the set rds of state ω, the set rds' of the next state ω' includes each (o, f) satisfying any of the following:

1. $(o, f) \in rds$, or
2. the current statement reads from field f of object o (that is, $stmt(\omega.p) \equiv y = x.f$ for any y and $\omega.\rho(x) = o$).

The abstract query NONSTATIONARYFIELD$^\alpha(f)$ is answered analogously to NONSTATIONARYFIELD(f), but replacing ω with ω^α, ρ with ρ^α, and rds with rds^α.

4. Abstractions

In this section, we present a family of heap abstractions that we will study. Abstractions in this family refine the classic allocation site abstraction along three dimensions: call stack, object recency, and heap connectivity.

Recall that an abstraction function α maps an object $o \in \mathbb{O}$ in a state $\omega \in \Omega$ to an abstract object $\alpha(\omega, o) \in \mathbb{A}$, where this mapping is computed using the appropriate auxiliary instrumentation $\omega.u \in \mathbb{U}$ (see Figure 2). The allocation site abstraction, denoted ALLOC, maps an object to the site where it was allocated:

$$\text{ALLOC}(\omega, o) = \omega.as(o). \quad (18)$$

Although very popular, the plain allocation site abstraction can be too coarse to prove many properties [17]. In the subsequent three sections, we will walk through the three dimensions of refinement, using a THREADESCAPE example as motivation (Figure 4).

4.1 Call Stack

Consider Example 1 in Figure 4. Variables `x` and `y` point to distinct objects, but because they are allocated at the same

```
    getnew() {
h1:   return new
    }
p2: x = getnew()
p3: y = getnew()
    spawn y
p1: ... x.f ...
```

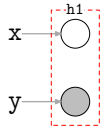
Example 1

```
    while (*) {
      x = new
p1:   ... x.f ...
      spawn x
    }
```

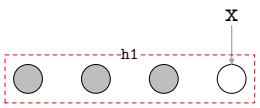
Example 2

```
h1: s = new
    spawn s
h2: x = new
    y = x
    while (*) {
h3:   z = new
      y.f = z
      if (x.f == y)
        s.f = z
      y = z
    }
    x = x.f
p1: ... x.f ...
```

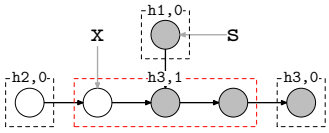
Example 3



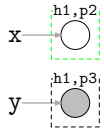
ALLOC



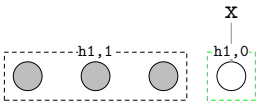
ALLOC$_{k=\infty}$



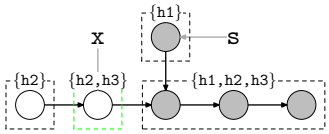
RECENCY



ALLOC$_{k=1}$



RECENCY



REACHFROM

Figure 4: Three examples that show the strengths and weaknesses of various abstractions. In each example, the goal is to prove that `x` is thread-local at `p1`, corresponding to the query at the field-accessing statement `... x.f ...`. The heap graphs at query time for two abstractions are shown below each code snippet. In Example 1, `x` is local but `y` escapes. Since both are allocated at `h1`, the allocation site abstraction (ALLOC) cannot prove `x` local, but ALLOC$_{k=1}$, which augments `h1` with the call site (`p2` and `p3`) can. In Example 2, no k value suffices to distinguish `x` from the rest, but object recency does differentiate the last object allocated from the rest. In Example 3, RECENCY is insufficient to distinguish `x` from the other elements of the linked-list, but REACHFROM can because the other elements are reachable from an additional allocation site `h1`.

allocation site (`h1`), the allocation site abstraction cannot distinguish between the two objects, and thus `x` cannot be proven thread-local at `p1`.

The most common way to refine this abstraction is to use k-CFA with heap cloning; let $\{\text{ALLOC}_k : k = 0, 1, 2, \dots\}$ denote these abstractions. Specifically, ALLOC$_k$ maps an object o to the allocation site of o ($\omega.as(o)$) and the k most recent call sites on the stack of the thread at the point at which o was allocated ($\omega.cs(o)[1..k]$):

$$\text{ALLOC}_k(\omega, o) = \langle \omega.as(o), \omega.cs(o)[1..k] \rangle. \qquad (19)$$

When $k = 0$, we recover the original allocation site abstraction. With call stack information, we can see that ALLOC$_{k=1}$ can make the relevant distinctions in Example 1 to prove `x` thread-local at `p1`.

The calling context is especially important in code where factory methods are frequently used, since in this case, many different objects are allocated at only one site and cannot be distinguished by ALLOC. Large k values might be especially important when such methods exist deep in heavily-reused code such as the JDK standard library. One of the goals of this paper is to study how large k must be in order to prove various queries.

We can also implement k-object sensitivity [20] in our framework by simply replacing the call sites in (19) with

the appropriate allocation sites, but we did not pursue this empirically.

4.2 Object Recency

In some cases, no amount of call stack information (even $k = \infty$) can help distinguish enough objects to prove a query. Consider Example 2 in Figure 4. The program repeatedly creates an object and renders it thread escaping. Therefore, at `p1`, all objects except for the most recent one are escaping. But since all objects have the same allocation site and call stack, even $\text{ALLOC}_{k=\infty}$ cannot prove `x` thread-local at `p1`.

This example therefore motivates refining allocation site abstractions to distinguish objects by their creation time. This is object recency idea, proposed by [4], which allows fine-grained reasoning about loops.

Recall that $\omega.os$ is the sequence of objects which have been allocated so far (in that order). For an object o (whose abstraction we're trying to compute), define the subsequence of $\omega.os$ which contains only objects with the same ALLOC_k abstraction as o:

$$relos_k(\omega, o) = \\ [o' : o' \in \omega.os, \text{ALLOC}_k(\omega, o) = \text{ALLOC}_k(\omega, o')]. \quad (20)$$

Now define the *recency index* of object o to be the position of o relative to the end of list $relos_k(\omega, o)$, truncated at r:

$$recidx_k(\omega, o) = \\ \min\{indexof(reverse(relos_k(\omega, o)), o), r\}. \quad (21)$$

We have $recidx_k(\omega, o) = 0$ if o is the last element of $relos_k(\omega, o)$, 1 if it is next to last, etc.

Finally, define abstraction RECENCY_k^r to map an object o to its ALLOC_k abstraction along with its recency index:

$$\text{RECENCY}_k^r(\omega, o) = \langle \text{ALLOC}_k(\omega, o), recidx_k(\omega, o) \rangle. \quad (22)$$

Note that when $r = 0$, we recover ALLOC_k. For $r > 0$, we can distinguish between $r + 1$ objects with the same allocation site abstraction. The only setting considered in past work is $r = 1$, so for convenience, we simply write RECENCY_k for $\text{RECENCY}_k^{r=1}$, RECENCY^r for $\text{RECENCY}_{k=0}^r$, and RECENCY for $\text{RECENCY}_{k=0}^{r=1}$.

Returning to Example 2, we see that RECENCY distinguishes between the last allocated object (which `x` points to) and the others, allowing us to prove the query thread-local.

The form of Example 2 is a common paradigm in server-like programs, where new objects are repeatedly constructed and subsequently released to other threads. The point is that during the construction phase, the objects are thread-local, but in order to prove this, one needs to distinguish the objects in the current loop iteration from the ones in previous loop iterations.

4.3 Heap Connectivity

Note that the RECENCY^r abstraction is heavily tied to single allocation sites; objects allocated at a site eventually collapse to the same abstraction after r objects are created. For programs that maintain complex data structures, objects allocated at one site might enter into a diverse set of relationships and have different properties over time. For these programs, more sophisticated shape analysis might be important.

We consider two kinds of abstractions, POINTEDTOBY_k [33] and REACHFROM_k [27], which combine shape predicates with k-CFA. Standard shape analysis predicates are based on local variable names [16, 27], which offer more distinctions than allocation sites. However, we use allocation sites instead of variable names in order to focus on the effect of adding shape information without conflating the contribution of variable names (which are another orthogonal dimension of refinement which warrants further investigation).

Specifically, POINTEDTOBY_k maps an object o to the set of allocation sites of objects which can reach o in *at most* one step:

$$\text{POINTEDTOBY}_k(\omega, o) = \\ \{\text{ALLOC}_k(\omega, o') : o' = o \vee o'.f = o\}. \quad (23)$$

Note this is a reflexive version of the pointed-to-by relation (the allocation site of o is always included in the set), which is non-standard. Similarly, REACHFROM_k maps an object o to the set of allocation sites of objects which can reach o in a finite number of steps:

$$\text{REACHFROM}_k(\omega, o) = \{\text{ALLOC}_k(\omega, o') : o' \overset{\omega.\sigma}{\rightsquigarrow} o\}. \quad (24)$$

Consider Example 3 in Figure 4. Here, a linked-list is created whose third node is rendered escaping. RECENCY cannot distinguish between the second node and any following node except the last. One could increase r to let RECENCY make more distinctions, but r would have to grow linearly with the list's length, rendering the approach impractical.

Turning to REACHFROM, note that the second node is reachable from allocation sites `h2` (the first node) and `h3` (the second node), while the third node onwards are in addition reachable from `h1`. Therefore, REACHFROM is able to separate the second node and deem it thread-local.

RECENCY and REACHFROM really capture different aspects of the heap—one does not strictly dominates the other. Which one works better depends on the benchmark and client involved, and thus for the remainder of the paper, we turn to an empirical study to provide more insight.

An implementation note: computing the REACHFROM abstraction is expensive since the abstraction of an object o depends on other objects in the heap, and thus each local heap update requires computing reachability information and updating the abstraction for a potentially large set of nodes. We use a dynamic data structure which maintains, for each object o, the set of objects that can reach o. We efficiently handle cases where a node is not on a cycle with any

of its immediate predecessors; various other optimizations are also employed.

A final remark: Note that POINTEDTOBY, REACHFROM, and RECENCY are state-dependent in that $\alpha(\omega, o)$ depends on ω. This dependence gives these abstractions more power, but also at some computational expense. In contrast, ALLOC_k for any k value is state-independent.

5. Benchmarks

We experimented with nine Java programs from the DaCapo benchmark suite (version 9.12) [5]:

`antlr`	A parser generator and translator generator
`avrora`	A simulation and analysis framework for AVR microcontrollers
`batik`	A Scalable Vector Graphics (SVG) toolkit
`fop`	An output-independent print formatter
`hsqldb`	An SQL relational-database engine
`luindex`	A text indexing tool
`lusearch`	A text search tool
`pmd`	A source-code analyzer
`xalan`	An XSLT processor for transforming XML

The suite provides three progressively larger inputs for each benchmark, via the "-s [small|default|large]" options, henceforth called SMALL, MEDIUM, and LARGE inputs, respectively. Due to computational constraints, we used SMALL inputs for our experiments. In Section 6, we show that the effect of the larger inputs on our conclusions is likely to be insignificant.

Table 1 provides various statistics of the benchmarks. The numbers refer only to classes, methods, and bytecodes that were visited during execution under SMALL inputs. The "app." columns provide numbers for application code (i.e., excluding the JDK standard library) while the "total" columns provide numbers for the entire code.

6. Experimental Setup

Our experiments were performed using IBM J9VM 1.6.0 on 32-bit Linux machines. We implemented all our abstractions and clients using Chord [1], an extensible static and dynamic program analysis framework for Java bytecode, built on top of the Joeq compiler infrastructure [34] and the Javassist bytecode instrumentation library [9]. Chord takes as input the class files, main entry point, and input data for each benchmark. Chord first runs the uninstrumented benchmark on the provided input data, and uses a lightweight JVM agent to observe all classes that are loaded, including both application and JDK library classes. It then instruments each class that was loaded (with the exception of `java.lang.J9VMInternals`) to generate an event whenever one of the following types of actions is executed in a method of that class:

- an object allocation (each `new` and `newarray`),
- a write to a static field of reference type (each `putstatic`),
- a read or write to an instance field or an array element of primitive or reference type (each `getfield`, `putfield`, `aload`, and `astore`),
- an explicit thread creation site (each call to the `start()` method of class `java.lang.Thread`),
- a lock acquisition site (each `monitorenter` as well as the entry point of each `synchronized` method), and
- the points immediately before and after method calls (each `invokevirtual`, `invokestatic`, etc.).

Each of these events is required by some abstraction (e.g., the pre- and post-method call events are required by k-CFA) or by some client (e.g., the `putstatic` event is required by the THREADESCAPE client to detect when objects become reachable from a static field).

Chord then runs the instrumented benchmark on the input data. Chord allows for the option of processing the generated events on-the-fly in a separate JVM with an uninstrumented JDK that communicates with the event-generating JVM via a POSIX pipe.[4] We did not employ this on-the-fly option as it produced non-deterministic traces for highly concurrent benchmarks. Instead, we ran the program once and wrote the trace of generated events to a binary file on disk. This allowed us to perform our analysis across different abstractions and clients on the same trace, yielding results which are meaningful to compare. However, saving to disk forced us to use the SMALL inputs for the DaCapo benchmarks because the MEDIUM and LARGE inputs resulted in enormous traces. The "# events generated" column in Table 2 shows the number of events generated on various input sizes; entries marked "?" denote that the experiment either ran for too long or ran out of memory.

While the number of reachable queries did generally increase for larger inputs as more application code became reachable, there was not much variation in the answers for the queries that were reachable under both SMALL and LARGE inputs. This observation is quantified in Table 2 for the THREADESCAPE client. The column "% change in reachable queries" has entries of the form $(-n, +m)$ meaning that the number of queries reachable under the indicated larger input but not under the SMALL input was $m\%$ of the queries reachable under the SMALL input; likewise, the number of queries reachable under the SMALL input but not under the indicated larger input was $n\%$. The most significant increases are for `antlr` (90%) and `batik` (38%).

The column "% change in true queries" is defined analogously with true queries instead of reachable queries. By this metric, even the most significant changes are quite small: 6% for `batik` and 4% for `pmd`.

[4] Using separate JVMs circumvents performance and correctness issues that would arise when event-processing code itself calls instrumented JDK libraries if one JVM were used.

benchmark	version	# classes		# methods		# bytecodes		# threads	
		app.	total	app.	total	app.	total	app.	total
antlr	2.7.2	89	290	845	1,663	102,426	147,774	1	5
avrora	cvs-20090612	399	678	1,726	2,882	89,397	161,925	4	8
batik	1.7	613	1,300	2,308	5,676	157,575	388,601	2	8
fop	0.95	868	1,357	4,467	6,764	373,657	511,713	1	5
hsqldb	1.8.0.4	111	465	1,013	2,597	103,879	212,472	42	46
luindex	2.4.1	170	495	1,019	2,453	73,527	161,152	1	5
lusearch	2.4.1	126	448	734	2,142	55,053	132,677	9	13
pmd	4.2.5	420	817	2,394	4,086	173,045	268,497	2	7
xalan	2.7.1	400	720	2,529	3,879	184,390	261,396	9	12

Table 1: Statistics of the DaCapo benchmarks used in this study.

benchmark	# events generated (in millions)			% change in reachable queries				% change in true queries			
	SMALL	MEDIUM	LARGE	MEDIUM		LARGE		MEDIUM		LARGE	
antlr	45.7	772	1,926	−0.4,	+90.1	−0.4,	+90.1	−0.0,	+0.0	−0.0,	+0.0
avrora	18.3	3,193	18,468	−0.0,	+0.2	−0.3,	+5.8	−0.0,	+0.0	−0.0,	+0.5
batik	59.4	572	731	−0.0,	+21.5	−0.0,	+38.2	−0.0,	+6.0	−0.0,	+6.0
fop	44.1	235	-	−5.9,	+5.2		-	−0.3,	+0.2		-
hsqldb	50.1	849	2,035	−0.0,	+0.3	−0.0,	+0.3	−0.0,	+0.0	−0.0,	+0.0
luindex	6.0	?	?		?		?		?		?
lusearch	16.6	2,184	4,662	−0.0,	+1.8	−0.0,	+1.8	−0.0,	+0.0	−0.0,	+0.0
pmd	13.2	940	?	−0.4,	+11.1		?	−0.0,	+4.2		?
xalan	29.6	2,219	?	−0.0,	+0.0		?	−0.0,	+0.0		?

Table 2: Trace lengths and variation in results for THREADESCAPE on different input data sets. A "-" means that the input size does not exist, and "?" means the experiment ran out of resources.

abstraction sub-family	refinements
$\{\text{ALLOC}_k\}_{k\in\mathbb{N}}$	k-CFA
$\{\text{RECENCY}_k^r\}_{k\in\mathbb{N},r\in\mathbb{N}}$	k-CFA, object recency to depth r
$\{\text{POINTEDTOBY}_k\}_{k\in\mathbb{N}}$	k-CFA, heap connectivity
$\{\text{REACHFROM}_k\}_{k\in\mathbb{N}}$	k-CFA, heap connectivity

Table 3: Abstractions we consider in this study.

7. Results

This section presents our empirical results on the family of abstractions considered in Section 4. Recall that we consider refinements of the basic allocation site abstraction (ALLOC) along three dimensions: k-CFA, object recency, and heap connectivity. Table 3 describes the abstractions we studied empirically.

For each abstraction, we obtain precision numbers for four clients and nine benchmarks. To navigate this large result space, we structure this section around the following questions:

- Independent of abstraction, what is the fraction of true queries (queries for which the answer is true) for a given client? (Section 7.1)
- Which abstraction works best for a given client? (Section 7.2)
- What is the effect of the k in k-CFA? (Section 7.3)
- What is the effect of the recency depth r? (Section 7.4)
- How scalable are the high-precision abstractions? (Section 7.5)

7.1 Client Statistics

Table 4 shows the queries which were true in the concrete execution for each benchmark and client (without abstraction). For the NONSTATIONARYFIELD client, the fraction of true queries is stable across benchmarks, but for the other three clients, this fraction varies considerably. In particular, variation in THREADESCAPE and SHAREDACCESS correlate with the amount and nature of concurrency in the benchmark program (Table 1).

Recall that THREADESCAPE measures reachability while SHAREDACCESS measures actual accesses. Indeed, we see the latter client has strictly fewer true queries, and moreover, the gap between the two clients is quite substantial for some benchmarks (notably `fop` and `xalan`), suggesting generally a higher use of static fields.

benchmark	ThreadEscape			SharedAccess			SharedLock			NonStationaryField		
	# true	# total	percent	# true	# total	percent	# true	# total	percent	# true	# total	percent
antlr	17	3490	0.5	0	3490	0.0	0	78	0.0	101	377	26.8
avrora	1983	5169	38.4	1755	5169	34.0	17	76	22.4	132	1037	12.7
batik	312	5028	6.2	0	5028	0.0	13	165	7.9	215	988	21.8
fop	3791	13734	27.6	0	13734	0.0	5	123	4.1	391	2114	18.5
hsqldb	1674	3817	43.9	1095	3817	28.7	41	135	30.4	157	576	27.3
luindex	139	3616	3.8	0	3616	0.0	5	160	3.1	202	637	31.7
lusearch	144	2490	5.8	51	2490	2.0	20	105	19.0	109	459	23.7
pmd	513	6345	8.1	45	6345	0.7	16	94	17.0	161	928	17.3
xalan	3702	7717	48.0	496	7717	6.4	43	116	37.1	291	1306	22.3

Table 4: For each benchmark (row) and client (column), we report the number of total queries issued ($|\mathbb{Q}|$), the number of queries for which the answer is true, and the corresponding percentage. For all clients except for SharedLock, only queries from application code are reported; for SharedLock, queries from the JDK standard library are also included because there are few locks in application code.

7.2 Effect of Abstraction on Clients

In this section, we focus on four abstractions (Alloc, Alloc$_{k=5}$, Recency, and ReachFrom), which allows us to explore the three dimensions of refinement independently. Tables 5–8 provide the precision results for the four clients on all nine benchmarks. Benchmark-client pairs where all queries are false are marked with "-" as a placeholder. A bold number indicates that it is within 1% of the precision of the best abstraction on that benchmark-client pair.

Let us start with the ThreadEscape client. From Table 5, we see that the plain allocation site abstraction is quite imprecise (average precision of 34.8%). Alloc$_{k=5}$ improves the precision significantly for the majority of the benchmarks (e.g., `fop`), but has little impact on others (e.g., `batik`). Recency is on average less effective than Alloc$_{k=5}$, though there are exceptions (e.g., `hsqldb`). ReachFrom performs slightly better than Recency.

The SharedAccess and SharedLock clients have similar behavior, as seen in Tables 6 and 7. In contrast to ThreadEscape, these two clients receive little improvement from Alloc$_{k=5}$. On the other hand, Recency is quite effective, outperforming or tying the other three abstractions uniformly across all benchmarks. ReachFrom seems to perform similarly to Alloc and Alloc$_{k=5}$, suggesting that these clients and benchmarks do not need sophisticated reasoning about the shape of the heap. Instead, the simple temporal notion captured by Recency seems to suffice.

For NonStationaryField, the case for Recency is stronger. From Table 8, we see that there is a huge gap between the precision of Recency (90.7%) and the other abstractions. Both Alloc and Alloc$_{k=5}$ perform equally poorly (40–50%). ReachFrom sits approximately half way in between. It is intuitive that Recency performs well on NonStationaryField, as this client is intrinsically built around a temporal property (writes must *precede* reads), and Recency focuses on making temporal distinctions.

benchmark	Alloc	Alloc$_{k=5}$	Recency	ReachFrom
antlr	48.6	85.0	81.0	**100.0**
avrora	54.7	62.3	69.2	**77.8**
batik	13.5	15.1	**20.9**	**20.6**
fop	36.3	**99.3**	42.8	41.3
hsqldb	62.6	69.0	**94.3**	?
luindex	6.3	**97.2**	6.8	6.8
lusearch	14.3	**90.0**	19.0	19.6
pmd	12.4	**87.1**	14.9	14.6
xalan	64.0	**78.9**	**78.7**	76.6
average	34.8	**76.0**	47.5	?

Table 5: Precision results for the ThreadEscape client.

benchmark	Alloc	Alloc$_{k=5}$	Recency	ReachFrom
antlr	-	-	-	-
avrora	96.2	96.2	**98.6**	?
batik	-	-	-	-
fop	-	-	-	-
hsqldb	51.3	56.8	**87.0**	?
luindex	-	-	-	-
lusearch	**3.5**	**3.5**	**3.6**	**3.5**
pmd	1.9	2.0	**18.4**	3.1
xalan	11.2	11.2	**15.0**	13.3
average	32.8	34.0	**44.5**	?

Table 6: Precision results for the SharedAccess client.

To conclude this section, there is a fair amount of variation in the precision of abstractions. However, two trends stand out: (1) Recency is a clear winner in three of the four clients, and in the exceptional ThreadEscape, k-CFA provides the most utility. So far, we have not seen ReachFrom to be very helpful, but we will see a case for ReachFrom

benchmark	ALLOC	$\text{ALLOC}_{k=5}$	RECENCY	REACHFROM
antlr	-	-	-	-
avrora	70.8	70.8	**85.0**	?
batik	**100.0**	**100.0**	**100.0**	81.2
fop	50.0	**100.0**	**100.0**	?
hsqldb	77.4	78.8	**91.1**	?
luindex	50.0	**100.0**	**100.0**	**100.0**
lusearch	29.9	32.8	**33.9**	32.3
pmd	40.0	43.2	**72.7**	40.0
xalan	55.1	55.1	**59.7**	58.1
average	59.1	72.6	**80.3**	?

Table 7: Precision results for the SHAREDLOCK client.

benchmark	ALLOC	$\text{ALLOC}_{k=5}$	RECENCY	REACHFROM
antlr	59.1	60.1	**91.0**	78.3
avrora	33.2	33.6	**93.6**	77.2
batik	35.8	36.1	**99.5**	65.3
fop	42.0	44.9	**90.9**	68.2
hsqldb	45.4	49.5	**94.6**	?
luindex	78.0	84.2	**94.8**	**94.8**
lusearch	38.2	38.2	**64.9**	56.5
pmd	37.8	39.9	**96.4**	69.4
xalan	44.0	44.5	**90.4**	74.2
average	45.9	47.9	**90.7**	?

Table 8: Precision results for the NONSTATIONARYFIELD client.

in Section 7.3. As mentioned in Section 4, shape analysis typically considers reachability from local variables rather than from allocation sites [16]. Since local variables generally offer finer distinctions than allocation sites, we expect a variable-based variant of REACHFROM to perform better. However, the use of variables is an orthogonal dimension (we could imagine RECENCY based on variables as well), which is outside the scope of this study.

7.3 Effect of k-CFA

In the previous section, we saw that $\text{ALLOC}_{k=5}$ was very useful for THREADESCAPE but not for the other clients. Is it because k needed to be higher? Could we have done just as well with $k < 5$ for THREADESCAPE? This section answers these questions.

Figure 5 plots the precision as a function of k for the four sub-families of abstractions in Table 3 (taking $r = 1$). First, consider the THREADESCAPE client (first row). We see that all abstractions work extremely poorly (precision around 20%) for small values of k, but there is a phase transition where the precision shoots up to nearly 100%. The critical value varies across abstractions and benchmarks, but is around $k = 3$ to $k = 6$. A partial explanation to the phase transition is as follows: Recall that THREADESCAPE is defined in terms of reachability, which is a sensitive property: a localized over-abstraction can render the entire subgraph downstream to be escaping.

Note that on the batik benchmark, even for $k = \infty$, both ALLOC_k and RECENCY_k fail to undergo the positive transition, but the heap connectivity abstractions (POINTEDTOBY and REACHFROM) do. On the remaining benchmarks, POINTEDTOBY and REACHFROM compare favorably with the other abstractions. After all, the THREADESCAPE client is defined in terms of reachability. In fact, we can redefine REACHFROM (24) to include a special label for static fields. This change actually has extremely positive consequences: the modified REACHFROM abstraction would always have 100% precision as it will never conflate objects reachable from static fields from those which are not.

On the other clients, we see that increasing k does not help in general (even with $k = \infty$). A notable exception is the pmd benchmark, where extremely large values of k make a significant difference. In a separate experiment on SHAREDACCESS, we saw that the precision did not reach its limit until $k = 18$. Further investigation is required to understand what properties of pmd make it an outlier, and in particular, which of its objects actually require a large k value.

In summary, the utility of k depends heavily on the client (THREADESCAPE versus others), and to a lesser extent on the benchmark (with the exception of pmd). Furthermore, we find that REACHFROM performs well on THREADESCAPE—it just takes larger k values to realize its potential, an instance of the synergy between two dimensions of refinement.

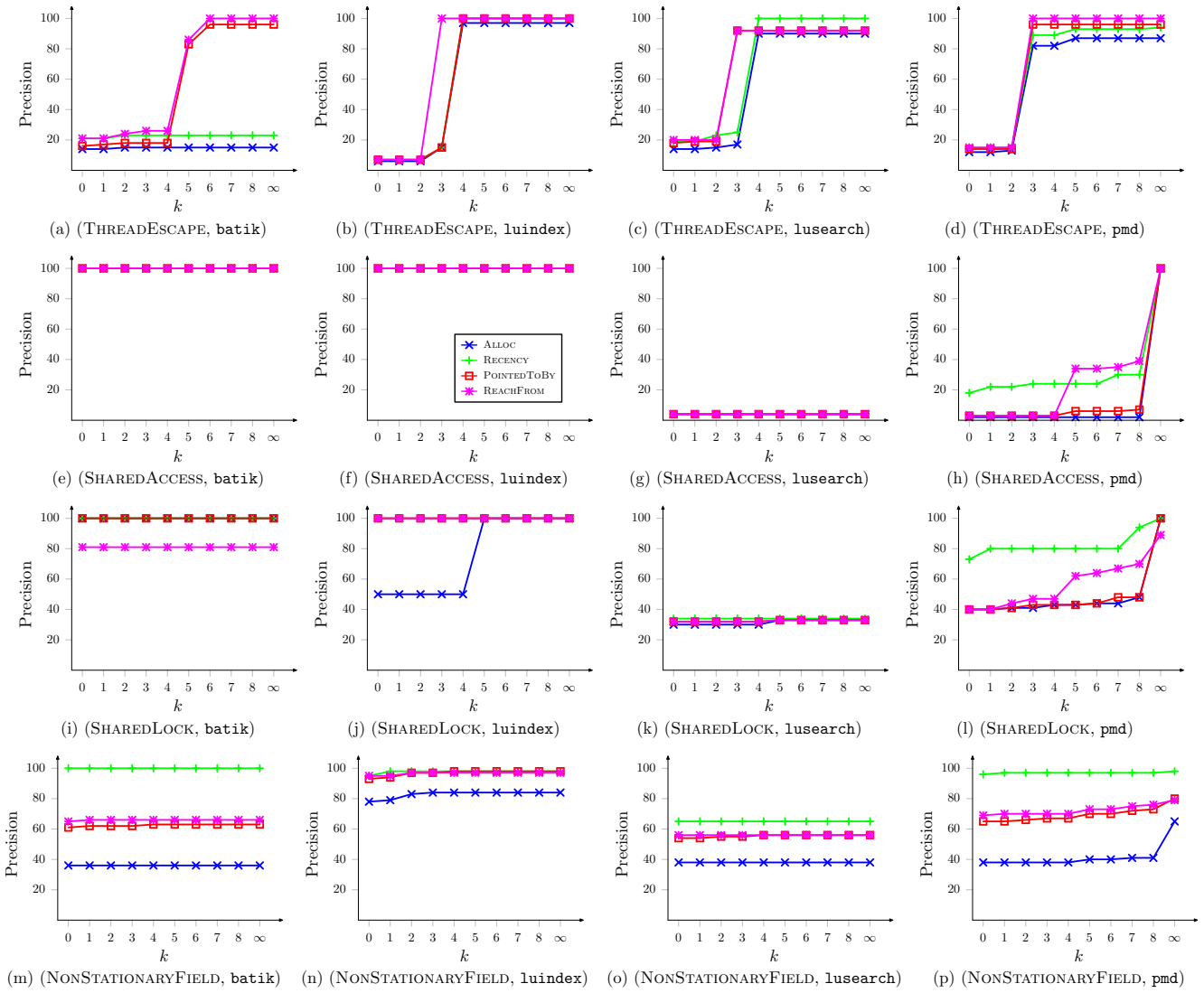
7.4 Effect of Recency Depth r

In Section 7.2, we saw that $\text{RECENCY}^{r=1}$ performed well. In this section, we investigate whether increasing the recency depth r adds any additional value.

We study two sub-families of abstractions, $\text{RECENCY}^{r}_{k=0}$ and $\text{RECENCY}^{r}_{k=\infty}$, for $0 \leq r \leq 6$.[5] The results are given in Table 9. We see that for most benchmark-client pairs, the precision increases by a fair amount as r increases until some point, larger values of r cease to be useful. An exception is lusearch, whose precision increases steadily up to $r = 6$ across all clients. In general, there seems to be a greater consistency across clients than for benchmarks, whereas for k-CFA, the effect was the opposite.

Finally, the gains from increasing k and increasing r are in general subadditive. However, one notable exception is THREADESCAPE on batik, where going from $r = 1$ to $r = 2$ when $k = 0$ increases precision by only 0.5% whereas going from $r = 1$ to $r = 2$ when $k = \infty$ increases precision by 75.6%. We saw a similar superadditive effect on the same benchmark-client pair in Section 7.3, where going

[5] Note that unlike $k = \infty$, $r = \infty$ trivially allows us to distinguish all objects and thus always achieve 100% precision.



(a) (ThreadEscape, batik) (b) (ThreadEscape, luindex) (c) (ThreadEscape, lusearch) (d) (ThreadEscape, pmd)

(e) (SharedAccess, batik) (f) (SharedAccess, luindex) (g) (SharedAccess, lusearch) (h) (SharedAccess, pmd)

(i) (SharedLock, batik) (j) (SharedLock, luindex) (k) (SharedLock, lusearch) (l) (SharedLock, pmd)

(m) (NonStationaryField, batik) (n) (NonStationaryField, luindex) (o) (NonStationaryField, lusearch) (p) (NonStationaryField, pmd)

Figure 5: Effect of varying k for various abstractions. For ThreadEscape, there is a sharp increase in precision around a critical value of k, but the precision does not depend much on k for the other clients (except on the pmd benchmark).

from Alloc to ReachFrom is useless if $k < 5$ but very useful if $k \geq 5$.

7.5 Tradeoff between Abstraction Precision and Size

Thus far, we have focused on evaluating the precision of abstractions. However, another important property of abstractions is how well they can scale inside a static analysis. To get at the notion of scalability in our dynamic analysis framework, we introduce the *size* of an abstraction α, which we define to be the number of abstract objects $|\mathbb{A}|$.

Figure 6 plots the precision versus size of various abstractions. Good abstractions live in the lower right-hand corner of the plot (exhibiting high precision with low complexity). As a baseline, we created a Random abstraction. To construct this abstraction, we fix a finite set of abstract values, $\mathbb{A} = \{1, \ldots, n\}$, and assign each object o independently to a random element of $\mathbb{A}$. We tried $n \in \{1000, 10000, 100000, \infty\}$, where $n = \infty$ corresponds exactly to the concrete result.

Random performs quite poorly on ThreadEscape. As mentioned earlier, ThreadEscape requires global reasoning about the heap, which is sensitive to arbitrary collapsing of concrete objects. On the other hand, for the NonStationaryField client, Random actually performs much better than the Alloc, PointedToBy and ReachFrom abstractions. This is an artifact of the client: when random objects of different types are collapsed, no information is actually lost with respect to stationarity because the two objects do not even have any fields in common.

		RECENCY$^r_{k=0}$							RECENCY$^r_{k=\infty}$						
		$r=0$	$r=1$	$r=2$	$r=3$	$r=4$	$r=5$	$r=6$	$r=0$	$r=1$	$r=2$	$r=3$	$r=4$	$r=5$	$r=6$
THRESC	batik	13.5	20.9	21.4	**22.1**	**22.5**	**22.6**	**22.7**	15.1	23.4	**99.0**	**99.0**	**99.0**	**99.0**	**99.0**
	luindex	**6.3**	**6.8**	**7.1**	**7.1**	**7.1**	**7.2**	**7.2**	97.2	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
	lusearch	14.3	19.0	22.4	22.7	23.0	23.2	**23.5**	90.0	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
	pmd	12.4	**14.9**	**14.9**	**14.9**	**15.0**	**15.0**	**15.0**	87.4	**93.6**	**93.6**	**93.6**	**93.6**	**93.6**	**93.6**
SHRDACC	batik	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	luindex	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	lusearch	3.5	3.6	3.9	5.3	5.6	6.7	**10.6**	3.5	3.6	4.0	5.5	6.0	7.6	**14.6**
	pmd	1.9	18.4	20.5	22.1	25.0	**29.6**	**29.6**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
SHRDLCK	batik	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
	luindex	50.0	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
	lusearch	29.9	33.9	35.7	40.0	40.8	46.5	**74.1**	32.8	33.9	35.7	40.8	41.7	47.6	**76.9**
	pmd	40.0	72.7	**80.0**	**80.0**	**80.0**	**80.0**	**80.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
NONSTFLD	batik	35.8	**99.5**	**99.5**	**99.5**	**99.5**	**99.5**	**99.5**	36.1	**99.5**	**99.5**	**99.5**	**99.5**	**99.5**	**99.5**
	luindex	78.0	94.8	**99.0**	**99.0**	**99.0**	**99.0**	**99.0**	84.2	**98.1**	**99.0**	**99.0**	**99.0**	**99.0**	**99.0**
	lusearch	38.2	64.9	73.6	87.9	88.6	**90.1**	**90.8**	38.2	65.3	79.0	93.2	**95.6**	**95.6**	**95.6**
	pmd	37.8	96.4	97.0	**97.6**	**97.6**	**98.2**	**98.2**	65.2	98.2	98.8	98.8	98.8	**99.4**	**99.4**

Table 9: Effect of increasing the recency depth r. Each cell shows the precision for the given RECENCY abstraction with a particular benchmark and client. Bold entries indicate precision within 1% of $r = 6$.

Note that as k increases, the size of the abstraction increases substantially, while the precision does not increase appreciably apart from the phase transitions. (Note that the Y-axis is on a log scale.) Shape-based abstractions (POINTEDTOBY and REACHFROM), though sometimes effective for THREADESCAPE, are quite costly because the abstract values are *sets* of allocation sites, which suffer from a combinatorial explosion. The number of abstract values may even exceed the number of concrete objects, since a single object may take on many abstractions during its lifetime as its connected heap changes.

Summary Although there is a fair amount of variation in precision across benchmarks and clients, this variation can be explained in terms of two main trends: First, the best abstractions for a client tend to correlate with the properties of the client: NONSTATIONARYFIELD involves temporal properties and thus benefits from RECENCY, which offers temporal distinctions; THREADESCAPE involves heap connectivity properties and thus benefits from REACHFROM. Second, there are non-trivial interactions between the various refinement dimensions (in one example, the potential of REACHFROM is only realized with k-CFA for large enough k). Overall, we showed that ALLOC is clearly insufficient, and RECENCY is a important dimension worthy of further exploration.

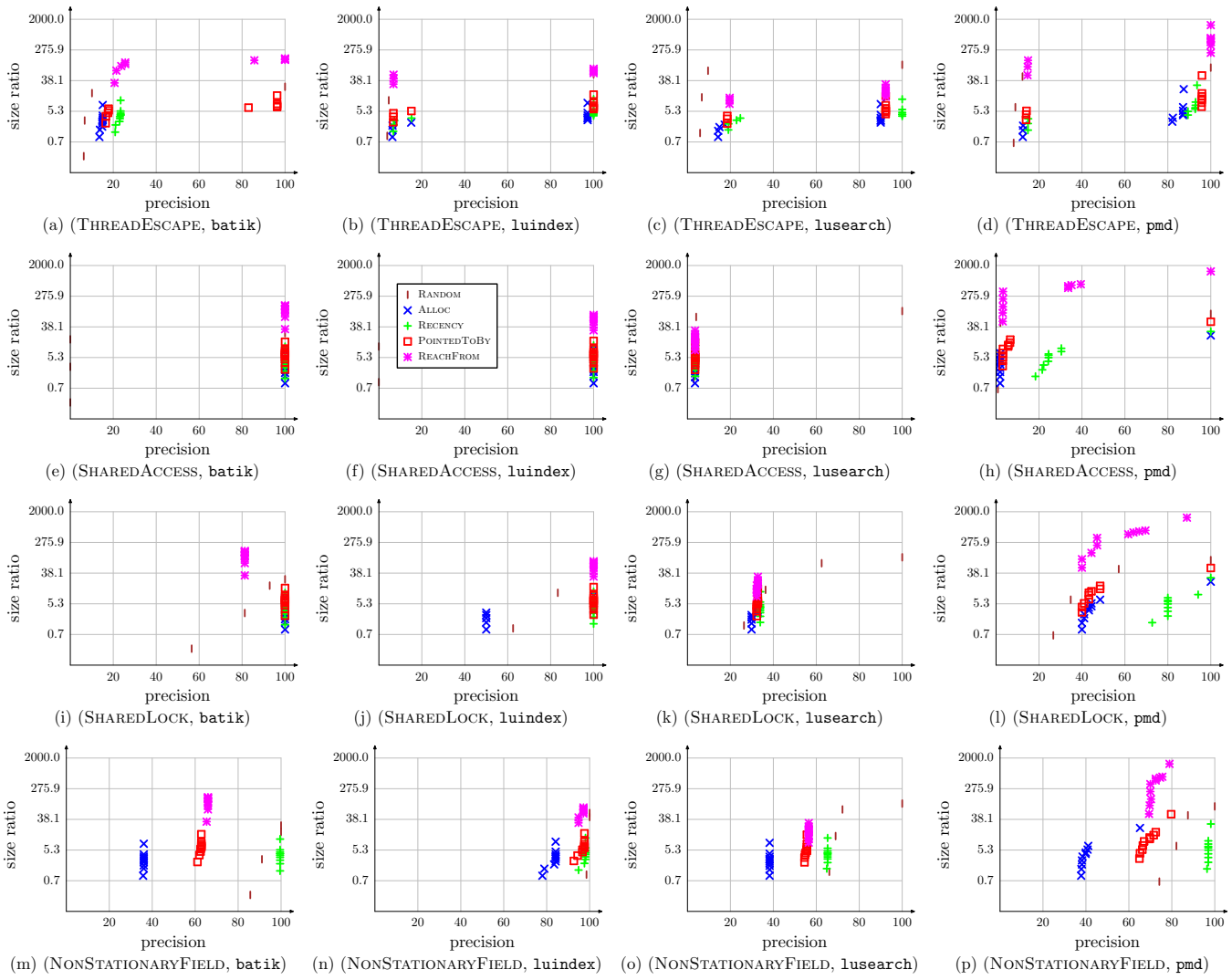
8. Related Work

A comprehensive presentation and evaluation of the k-CFA heap abstraction, as well as other k-limited abstractions like k-object-sensitivity, is presented in [17]. The recency abstraction was introduced in [4]. Shape analysis, which is a static technique for verifying properties of dynamically-allocated data structures, is presented in [27]. Here, we survey work on evaluating the impact of k-limited heap abstractions on points-to and call-graph algorithms (Section 8.1) and work on using connectivity-based heap abstractions to improve garbage collectors (Section 8.2) and detect memory bloat (Section 8.3).

8.1 Points-to and Call Graph Algorithms

Liang *et al.* [19] present a set of empirical studies investigating the effect of calling contexts on the precision of Andersen's algorithm. In particular, the effect of context-sensitive naming schemes on precision is evaluated, and the traditional calling context sensitivity is compared to object context sensitivity. The precision of the points-to information computed by each of the algorithms is evaluated *vis-a-vis* dynamically-collected data. In an earlier and closely related study [18], the authors perform a similar set of experiments (again, using reference information collected at runtime as an approximation of precise reference information), concluding that hybrid approaches for identifying instances and computing points-to information are needed.

Lhoták and Hendren [17] follow a similar approach. They conduct an empirical study on a set of large Java benchmarks to evaluate the precision of subset-based points-to analysis under three variations of context sensitivity: call string, object sensitivity, and the BDD-based context-sensitive algorithm proposed by Zhu and Calman, and by Whaley and Lam. They evaluate the effects of these variations on the number of contexts generated, the number of distinct points-to sets constructed, and the precision of call-graph construction, virtual-call resolution, and cast-safety analysis. Our study is complementary to theirs: we measure the effect of k-CFA on the precision of clients for much higher values of k, but for a single execution, whereas they measure the same for smaller values of k but over all executions. Some



(a) (ThreadEscape, batik) (b) (ThreadEscape, luindex) (c) (ThreadEscape, lusearch) (d) (ThreadEscape, pmd)

(e) (SharedAccess, batik) (f) (SharedAccess, luindex) (g) (SharedAccess, lusearch) (h) (SharedAccess, pmd)

(i) (SharedLock, batik) (j) (SharedLock, luindex) (k) (SharedLock, lusearch) (l) (SharedLock, pmd)

(m) (NonStationaryField, batik) (n) (NonStationaryField, luindex) (o) (NonStationaryField, lusearch) (p) (NonStationaryField, pmd)

Figure 6: Tradeoff between abstraction precision and size ratio for two clients and four benchmarks. (Note the logarithmic scale.) The size ratio is size of the abstraction $|\mathbb{A}|$ divided by the size of the Alloc abstraction. Multiple points for a given abstraction indicate increasing values of k which monotonically increases both precision and size. Note that the upper-rightmost Random point corresponds to the concrete execution.

abstractions we consider are different from theirs (e.g., they do not evaluate heap connectivity and we do not evaluate k-object sensitivity). Finally, all our clients are motivated by concurrency and are different from theirs.

8.2 Garbage Collection

Hirzel and Hind [13] explore whether the connectivity of objects can yield useful partitions or improve existing partitioning schemes. They consider direct points-to relations, as well as transitive reachability relations, and conclude that connectivity correlates strongly with object lifetimes, and is therefore useful for partitioning objects.

Shaham *et al.* [30] study the potential impact of different kinds of liveness information on the space consumption of a program in a garbage-collected environment. They measure the time difference between the actual time an object is reclaimed by the garbage collector and the earliest time in which this could have been done assuming the availability of liveness information. Four kinds of liveness information are considered: stack reference, global reference, heap reference, and any combination of the above.

Inoue *et al.* [14] introduce a precise method for predicting object lifetimes, where the granularity of predictions is equal to the smallest unit of allocation. They construct predictors based on execution traces including accurate records of each object's allocation and death, and rely on a (fixed-length) prefix of the stack at the time of allocation to disambiguate allocation contexts and thus improve the precision of the pre-

dictor. Their empirical results suggest that for some applications, object lifetimes can be predicted to the byte using the allocation-context heuristic. This finding resonates well with our empirical results, which show that the THREADESCAPE client benefits greatly from high k values.

The approach taken in [14] is inspired by the work of Seidl and Zorn [28, 29], which attempts to predict the reference and lifetime behavior of heap objects according to four categories: highly referenced, short lived, low referenced, and other. Similar to [14], prediction relies on a training trace containing extensive information, including the number of loads and stores to each object, the call stack at the time of each allocation, and the size of the allocated object. Seidl and Zorn use a stack-based prediction scheme, and conclude that it is important to choose the right depth for the stack predictor. Their experiments suggest that a depth of 3 yields an effective predictor.

While all four of these studies provide insightful observations that may be leveraged by a static analysis, their goal differs from ours. Our goal is to evaluate static abstractions explicitly, so as to provide insight to static-analysis developers, whereas these studies focus on (concrete) empirical results, and establish heuristics that may benefit a garbage collector.

8.3 Memory Bloat

Mitchell [21] investigates ways to summarize the memory footprint of object-oriented applications in order to discover cases where high-overhead collections, bulky data models and large caches are used. As part of the analysis, he develops a catalog of ownership structures, which are shown to be prevalent in large-scale applications, and are therefore powerful units of aggregation and filtering. In a related study, Mitchell and Sevitsky [22] study applications posing large runtime memory requirements. They introduce "health signatures" to distinguish cases where a large memory footprint enables an important requirement (e.g., the use of a cache to ameliorate a performance problem) from instances where memory is used excessively.

Also closely related is Yeti [23], a tool for summarizing memory usage to uncover the costs of design decisions. This is accomplished through a series of progressive abstractions and corresponding visualizations. The goal behind Yeti is to assist developers in discovering instances where large-scale Java applications suffer from memory problems due to an inefficient design, or lifetime bugs such as leaks.

Similar to our study, these works are concerned with abstracting the concrete heap. However, the abstractions advocated by these studies are not inspired by static analysis; rather, they are more heuristic in nature, and are tuned toward an interactive process wherein "simplification" of the concrete heap is beneficial (indeed mandated) as part of a reasoning process leading to the identification of evasive bugs and design flaws.

Dufour *et al.* [11] introduce *blended analysis*, an algorithm combining a dynamic representation of the program's calling structure with a static analysis applied to a region of that calling structure with observed performance problems. In a case study they perform, they show that blended escape analysis is highly effective at localizing a performance problem due to overuse of temporary structures. In a subsequent study [12], new metrics are added to quantify key properties of temporary data structures and their uses, and an empirical evaluation is conducted to characterize temporaries in framework-intensive applications.

Dufour's studies are similar to our work in that a static representation of the program is computed on top of a dynamic trace. However, whereas Dufour's focus is on localizing a particular problem or behavior (which is present during the concrete execution), our goal is to understand which heap abstractions are likely to be good for static analysis.

9. Conclusion

With the goal of finding good heap abstractions for static analysis, we have investigated a family heap abstractions on four clients and nine benchmarks. Our evaluation of these abstractions revealed many interesting properties about the role and utility of k-CFA, object recency and heap connectivity. We believe these results can serve as a useful guide for developing static analyses.

References

[1] Chord: A static and dynamic program analysis framework for Java. `http://code.google.com/p/jchord/`.

[2] J. Aldrich, C. Chambers, E. G. Sirer, and S. J. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the 6th Intl. Static Analysis Symp. (SAS)*, pages 19–38, 1999.

[3] J. Aldrich, E. Sirer, C. Chambers, and S. J. Eggers. Comprehensive synchronization elimination for Java. *Science of Computer Programming*, 47(2-3):91–120, 2003.

[4] G. Balakrishnan and T. W. Reps. Recency-abstraction for heap-allocated storage. In *Proceedings of the 13th Intl. Static Analysis Symp. (SAS)*, pages 221–239, 2006.

[5] 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 *Proceedings of the 21st ACM SIGPLAN Conf. on Object-Oriented Programing Systems, Languages and Applications (OOPSLA)*, pages 169–190, 2006.

[6] B. Blanchet. Escape analysis for Java: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, 2003.

[7] B. Blanchet. Escape analysis for object-oriented languages: Application to Java. In *Proceedings of the 14th ACM SIG-*

PLAN Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pages 20–34, 1999.

[8] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the 14th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 35–46, 1999.

[9] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *Proceedings of the 2nd Intl. Conf. on Generative Programming and Component Engineering (GPCE)*, pages 364–376, 2003.

[10] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 258–269, 2002.

[11] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the ACM SIGSOFT Intl. Symp. on Software Testing and Analysis (ISSTA)*, pages 118–128, 2007.

[12] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *Proceedings of the 16th ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering (FSE)*, pages 59–70, 2008.

[13] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *Proceedings of the Workshop on Memory Systems Performance (MSP) and the Intl. Symp. on Memory Management (ISMM)*, pages 143–156, 2003.

[14] H. Inoue, D. Stefanovic, and S. Forrest. On the prediction of java object lifetimes. *IEEE Transactions on Computers*, 55: 880–892, 2006.

[15] K. Lee and S. P. Midkiff. A two-phase escape analysis for parallel Java programs. In *Proceedings of the 15th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 53–62, 2006.

[16] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *In Intl. Symp. on Software Testing and Analysis*, pages 26–38, 2000.

[17] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In *Proceedings of the 15th Intl. Conf. on Compiler Construction*, pages 47–64, 2006.

[18] D. Liang, M. Pennings, and M. J. Harrold. Evaluating the precision of static reference analysis using profiling. In *Proceedings of the ACM SIGSOFT Intl. Symp. on Software Testing and Analysis (ISSTA)*, pages 22–32, 2002.

[19] D. Liang, M. Pennings, and M. J. Harrold. Evaluating the impact of context-sensitivity on andersen's algorithm for java programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, pages 6–12, 2006.

[20] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the ACM SIGSOFT Intl. Symp. on Software Testing and Analysis (ISSTA)*, pages 1–11, 2002.

[21] N. Mitchell. The runtime structure of object ownership. In *Proceedings of the 20th European Conf. on Object-Oriented Programming (ECOOP)*, pages 74–98, 2006.

[22] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *Proceedings of the 22nd ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 245–260, 2007.

[23] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In *Proceedings of the 23rd European Conf. on Object-Oriented Programming (ECOOP)*, pages 77–97, 2009.

[24] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 327–338, 2007.

[25] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 308–319, 2006.

[26] E. Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 208–218, 2000.

[27] M. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[28] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. *SIGOPS Oper. Syst. Rev.*, 32 (5):12–23, 1998.

[29] M. L. Seidl, M. L. Seidl, M. L. Seidl, B. G. Zorn, B. G. Zorn, and B. G. Zorn. Predicting references to dynamically allocated objects. Technical report, 1997.

[30] R. Shaham, E. K. Kolodner, and M. Sagiv. Estimating the impact of heap liveness information on space consumption in Java. In *Proceedings of the Workshop on Memory Systems Performance (MSP) and the Intl. Symp. on Memory Management (ISMM)*, pages 171–182, 2002.

[31] O. Shivers. Control-flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 164–174, 1988.

[32] C. Unkel and M. S. Lam. Automatic inference of stationary fields: a generalization of Java's final fields. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 183–195, 2008.

[33] E. Y.-B. Wang. *Analysis of Recursive Types in an Imperative Language*. PhD thesis, Univ. of Calif., Berkeley, CA, 1994.

[34] J. Whaley. Joeq: A virtual machine and compiler infrastructure. *Science of Computer Programming*, 57(3):339–356, 2005.

[35] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *Proceedings of the 20th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 265–274, 2008.