PHALANX: Parallel Checking of Expressive Heap Assertions

Greta Yorsh
IBM Research

Eran Yahav
IBM Research

Martin Vechev
IBM Research

Bard Bloom
IBM Research

Abstract

Unrestricted use of heap pointers makes software systems particularly difficult to understand. Incidental and accidental pointer aliasing result in unexpected side effects of seemingly unrelated operations, and are a major source of system failures. Such failures are hard to test or debug with existing tools, especially for concurrent programs.

We present PHALANX — a practical framework for dynamically checking expressive heap properties such as ownership, sharing and reachability. PHALANX uses novel parallel algorithms to efficiently check heap properties utilizing the available cores in the system.

To debug her program, a programmer can annotate it with expressive heap assertions in JML, which use heap primitives provided by PHALANX. The framework combines a modified version of the JML compiler with a specialized runtime to efficiently evaluate these assertions using parallel algorithms. The PHALANX runtime has been implemented on top of a production virtual machine.

We applied PHALANX to real world applications in various scenarios, and found expressive heap assertions to be extremely valuable in debugging and program understanding. Further, our experimental results indicate that evaluating heap queries using parallel algorithms can lead to significant performance improvements, often resulting in linear speedups as the number of cores increases.

1. Introduction

Unrestricted use of heap pointers makes software systems particularly difficult to understand. Incidental and accidental pointer aliasing result in unexpected side effects of seemingly unrelated operations, and are a major source of system failures.

Despite significant advances in checking and verification of heap properties, it is still extremely challenging to check them in real-world applications. Static approaches are either imprecise (e.g., [3, 34]), do not scale to large applications (e.g., [33, 13, 35, 37]), or focus on specific properties (e.g., [9]). Most of these approaches do not handle concurrency. Dynamic approaches are more promising in terms of scaling, but are either limited to local properties [10], require heap snapshots to be analyzed offline [25], or support only a limited set of simple heap queries [4, 1]. Specification languages for Java such as JML [21] allow the programmer to specify expressive heap assertions, but lack runtime support for checking them.

It is particularly challenging to check heap properties that involve sharing, ownership, and transitive reachability. Their evaluation might require multiple traversals of the entire heap, and result in a prohibitive runtime. To alleviate this worst case behavior, we identify common heap queries, and provide a runtime environment that can leverage available system cores for evaluating these queries in parallel.

With the advent of multicore systems, we envision that some cores could be dedicated to performing software quality tasks, and in particular for checking properties that have been traditionally considered too expensive even for debugging scenarios. This work is a step in that general direction.

We present PHALANX — a practical tool for dynamically checking a wide range of expressive heap assertions. To make assertion checking practical, PHALANX provides: (i) a small set of natural primitives that can be used in JML assertions for reasoning about the heap, and (ii) parallel algorithms for checking these assertions.

Expressive Language of Heap Queries The common heap queries we support are motivated by real usage scenarios, in which they proved to be valuable. Many of these scenarios reflect challenges with common programming patterns, including ownership and aliasing control, resource management, and event handling mechanisms. Our common queries enable the programmer to check properties that are difficult or impossible to check otherwise, such as properties that conceptually require traversing pointers backwards.

Our experience indicates that the heap queries required in practice are sometimes a significant refinement of the naive queries one would write. For example, as demonstrated in Section 2, practical queries may require reasoning about *reachability through paths that avoid a certain set of objects*, a property that cannot be phrased as a simple reachability query. Similar adaptations are required for making other queries useful in practice, and PHALANX provides parallel implementations for these refined queries.

Evaluating a heap query requires it to hold pointers to the relevant objects in the heap. The user-intended meaning of the query might not want to take such pointers into account. For example, when a user writes a query to check whether an object is shared, the query itself must hold a pointer to the object, which the user probably does not want to account for as a source of sharing (otherwise, unless the query holds the only pointer to the object, the query will always return true, by definition). To clarify such subtleties, we define a formal semantics for our heap queries. This also helps us in proving that our parallel algorithms compute the intended results.

To make the deployment of common heap queries easy, we provide a small set of simple primitives that can be used in JML assertions for reasoning about the heap. Common heap queries can be naturally expressed in JML using these new primitives, together with quantifiers and set operations available in JML. We use a modified version of the JML compiler to map these queries to their parallel implementation in the PHALANX runtime.

Efficient Evaluation of Heap Queries To provide efficient checking of heap assertions, our implementation is based on a production virtual machine (VM). We use and adapt existing components of the VM, combining them with novel parallel algorithms. Having a parallel implementation inside the VM achieves efficiency that is very hard to achieve by any other means.

Since PHALANX is based on a production VM, we can use it to run real-world applications. While adding meaningful heap queries to such applications requires intimate knowledge of the code, we experimented with several heap queries that are of general applicability. Using these queries we found a small number of potential sources of bugs and inefficiencies.

While the parallel garbage collector (GC) inside the VM provides us with some basic components, certain heap queries, e.g., involving domination and disjointness, cannot be checked in a single GC marking traversal. One of the main contributions of this paper is novel parallel algorithms that are designed to efficiently check common heap queries.

1.1 Main Contributions

The contributions of this paper include:

- A set of common heap queries pertaining to global properties of the heap (e.g., ownership, sharing, and reachability), and usage scenarios where we found these common heap queries to be useful.
- A small set of natural primitives that can be used to express heap queries inside JML assertions, making the use of heap queries easy and accessible.
- A modified JML compiler that maps common heap queries to efficient implementations in the PHALANX runtime.
- New parallel algorithms for efficient evaluation of the common heap queries, *and* implementation of the parallel algorithms in a production virtual machine.
- Experimental evaluation of heap queries on synthetic benchmarks and real-world applications, including comparison of the parallel implementation of the PHALANX runtime on top of QVM to a reference implementation based on JVMTI.

2. Motivating Example

In this section, we provide a simple motivating example for the use of heap assertions, and explain the meaning of some assertions in an informal manner. A more formal treatment is provided in Section 3.

Fig. 1 shows a code fragment from JdbF, a system for storing and retrieving objects in a relational database. The `Database` class provides an interface to clients of JdbF for performing various operations on the database. Each operation acquires a connection, performs its task on the database, and releases the connection. The `ConnectionManager` class maintains a map of all available connections. Each `Connection` object is confined in a `ConnectionSource` object. The invariant of the JdbF library is that every `Connection` is used by at most one database operation at a time.

A race in the original program, first reported in [28], violates this invariant. Since `getConnection` methods are not synchronized, two threads can concurrently pass the `!used` guard on line 27 and wind up with the same `Connection` c. An example of such memory configuration is shown in Fig. 2, where the connection object in the middle is shared by two running threads.

Instead of resorting to various methods for race-detection, the programmer can detect that her code violates the invariant. In our example, the programmer may want to check that *a connection object is reachable from at most one thread.*

```
1  public class Database {
2   private ConnectionManager cm;
3   // @ private invariant  \forall Connection c;true;
4   // @ (\num_of Thread t; Phalanx.running().has(t);
5   // @ Phalanx.reach(t, cm.conns.values()).has(c))<=1
6   public int insert(...) throws MappingEx {
7    Connection c = cm.getConnection(...);
8    ...
9   }
10  ...
11 }
12 public class ConnectionManager {
13  private /*@ spec_public @*/ Map conns =
14     Collections.synchronizedMap(new HashMap());
15  public Connection getConnection(String s)
16   throws MappingException {
17   try {
18    ConnectionSource c = conns.get(s);
19    if (c != null) return c.getConnection();
20    throw new MappingException(...);
21   } catch (SQLEx e) { ... }
22  }
23 }
24 public class ConnectionSource {
25  private Connection conn;
26  private boolean used;
27  public Connection getConnection() throws SQLEx {
28    if (!used) {
29     used = true;
30     return conn;
31   }
32   throw new SQLEx(...);
33  }
34 }
```

Figure 1. Code fragment from JdbF.

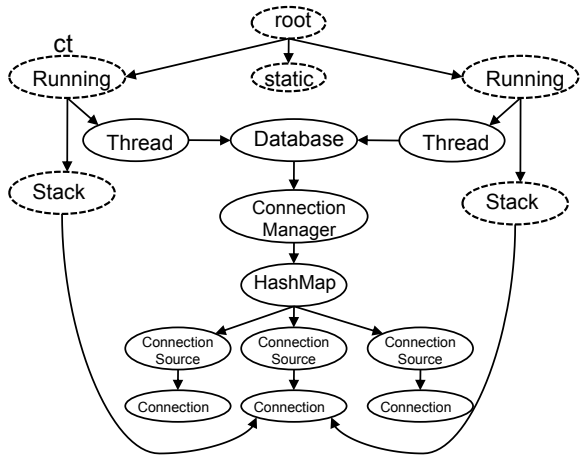

Figure 2. Example graph that represents a state of the program JdbF from Fig. 1. We use dashed line for the special nodes, which do not represent heap-allocated objects. All other nodes represent objects labeled with their (short) class name. We omit some details of the `Map` implementation for clarity.

This property can be expressed in JML by `(\num_of Thread t; t.isAlive(); \reach(t).has(c))<=1`.

The quantifier `\num_of` returns the number of thread objects t that satisfy both assertions `t.isAlive()` and `\reach(t).has(c)`. The primitive `\reach`, built-in in JML, returns the set of all objects reachable from the object referenced by t. The method call `has(c)` returns true when the `Connection` object c is in the set.

It is worth noting that efficient evaluation of the above heap query is nontrivial. The query iterates over the set of executing threads and checks (transitive) reachability from each one of these threads. Indeed, the JML compiler treats it as a non-executable assertion.

PHALANX can efficiently evaluate this assertion, using parallel heap traversal to implement the reachability check `\reach(t).has(c)`. PHALANX also provides a new primitive `running`, which returns the set of all executing threads. This primitive is easy to implement efficiently in PHALANX runtime, which has access to the VM's internal information. Finally, PHALANX evaluates this assertion atomically, i.e., all subexpressions are evaluated on the same snapshot of the heap and the application threads are suspended during the evaluation. Otherwise, a thread may be alive when we begin evaluating the assertion, but dead by the time we finish the iteration; or a thread may be modifying fields of objects while the assertion is being evaluated, thus altering reachability.

Even if the clients of JdbF are synchronized, the above assertion fails. In fact, all connection objects are reachable from all client threads, through the connection manager's connection map, as we see in Fig. 2. Note that such path properties are more complex than reachability.

Path properties cannot be expressed in JML in a natural way. Fortunately, JML language is designed to be easily extensible: new primitives are simply calls to (pure) methods implemented elsewhere. PHALANX uses this capability to provide primitives that capture path properties, such as reachability through or avoiding certain objects, and domination.

Using the primitives `running` and `reach` provided by PHALANX, a correct invariant can be expressed in JML, as shown in Fig. 1(lines 3-4).

In Section 4, we describe a number of parallel algorithms for evaluating such queries. The basic idea behind these algorithms is to parallelize the multiple transitive traversals required to evaluate certain queries. For most queries, this requires sophisticated coordination between heap-traversing threads.

3. Expressive Language of Heap Queries

JML assertion language [21] supports quantifiers and set operations; extended with a few primitives about the heap, it gives us a natural way to specify expressive heap properties. Additionally, JML assertion language allows us to write rich assertions that combine reasoning about the heap with

reasoning about other aspects of the program, and we benefit from the JML compiler support for method specification, inheritance of specification, invariant evaluation, etc.

In this section, we start by formally defining the meaning of heap queries. We present the primitives supported by PHALANX and their semantics in Table 1. Common heap queries that arise in many real-world usage scenarios can be written using the new primitives together with standard quantifiers and set operations, as shown in Table 2.

3.1 Semantics of Heap Queries

In this section, we define the meaning of heap queries in terms of Java program state. Towards this end, we define an abstraction function mapping concrete program states to their representation as graphs, and then define the meaning of queries on these graphs. It makes the semantics more accessible. It also simplifies the reasoning about correctness of our parallel implementation of the common heap queries, described in Section 4.

In Java operational semantics (e.g., see [23]), a program state consists of a global program counter, a current thread register, a heap, a thread table, an internal class table that records the runtime representations of the loaded classes, an environment that represents the source from which classes are to be loaded, and error flags.

To define the meaning of heap queries, we consider a program state that consists of: information about the heap, the thread table, and the internal class table. The heap is a map from addresses to objects. The thread table contains, for each running thread, the program counter and the method invocation stack (a.k.a. "call stack"). The call stack is a list of frames; a frame includes the method being invoked, its operands, local variables, and an object on which the invocation is synchronized. The internal class table is a map from class names to direct superclass, interfaces, fields, methods, access flags, and byte code.

Graph definition $g(s)$***:*** For a state s as above, we define a directed graph $g(s)$ whose nodes include the heap-allocated objects labeled by types, and whose edges represent references between objects. The meaning of heap queries is defined by graph- and set-theoretic operations on $g(s)$.

EXAMPLE 3.1. *Consider a state of the JdbF program of Fig. 1, in which two threads concurrently execute* `insert` *on the same* `Database` *object. Fig. 2 shows the corresponding graph. In this graph, nodes with solid boundary lines represent heap allocated objects. Nodes with dashed lines represent additional information such as threads runtime information and thread stacks. In the graph of Fig. 2, the database contains three elements of* `ConnectionSource`*, each of which has a field pointer to its* `Connection` *object. Each thread has a stack pointer to a* `Connection` *object, via local variable* `c`*. Note that for each thread, this stack pointer goes from the corresponding stack node in the graph. Each thread also has a field pointing to the shared* `Database` *object. Note that this heap pointer goes directly from the* `Thread` *object to the* `Database` *object. Every* `Connection` *object is transitively reachable from all running threads through the* `Map` *object. Furthermore, both running threads refer to the same* `Connection` *object from their stack.*

Formally, given a state s we define $g(s)$ to be a tuple $\langle V, E, L\rangle$ where V is the set of nodes, $E \subseteq V \times V$ is the set of edges, and $L\colon V \rightarrow \mathit{Classes} \cup \{\mathit{root}, \mathit{stack}, \mathit{running}, \mathit{static}\}$ is the node-labeling function. The graph $g(s)$ is defined as follows.

Every allocated object in the heap of s is represented by a node in the graph. The node is labeled with the dynamic type of the object, from *Classes*. The set of nodes that represent heap objects in s is denoted by $\mathit{heap}_{g(s)} \subseteq V$. For every object o and every non-primitive, non-null field f of the object's dynamic type, including private and protected fields, and including all those defined in superclasses, there is an edge in the graph from the node representing o to the node representing the value of the field f. Every thread object is represented in the graph by a node, labeled with its type, just like any other object. If the thread is running, the graph will include two more nodes: a node labeled by *running*, which represents the thread's runtime information, and a node labeled by *stack*, which represent the thread's call stack. The set of *running* nodes is denoted by $\mathit{running}_{g(s)} \subseteq V$. The set of nodes that represent call stacks is denoted by $\mathit{stack}_{g(s)} \subseteq V$. There are edges from the node that represents a running thread (labeled with *running*) to the node that represents its call stack and to the node that represents its thread object. For a node $r \in \mathit{running}_{g(s)}$, we use $r.\mathit{stack}$ and $r.\mathit{obj}$ to denote the corresponding stack and thread-object nodes. For every thread, for every method on the thread's call stack, and for every non-null, non-primitive local variable of the method, including formal parameters, there is an edge in the graph from the *stack* node that represents the thread's call stack to the node that represent the value of the local variable. Note that a single node represents all the stack frames together. As $g(s)$ is not a multigraph, there is a single edge from a stack node to an object referenced from the stack, even if many stack variables refer to the same object.

The graph contains a designated node, labeled by *static*. For every loaded class, and every static non-primitive non-null field in the class, including private and protected fields, there is an edge from the node labeled with *static* to the node that represents the value of the field. The graph also contains a designated node, labeled with *root*. For every running thread, there is an edge from the *root* node to the node that represents the running thread. Additionally, there is an edge from the *root* node to the *static* node. Finally, the node $ct \in \mathit{running}_{g(s)}$ refers to the currently-running thread.

Graph Operations We now introduce some standard notations for the graph operations which we use in the definition of the semantics. Let $g(s) = \langle V, E, L\rangle$ be a graph as above.

For a node $v \in V$, we use $pred_{g(s)}(v) \subseteq V$ to denote the predecessors of v. A path in $g(s)$ is a (non-empty) sequence of nodes $v_1, \ldots, v_m$ such that for all $i = 1, \ldots, m-1$, $(v_i, v_{i+1}) \in E$. We use $src(\pi)$ and $dst(\pi)$ to denote the first and last nodes of the path π. For a path π, we use $nodes(\pi)$ to denote the set of nodes on the path, including the endpoints. The set of all paths in the graph $g(s)$ is denoted by $\Pi_{g(s)}$. We use $\Pi_{g(s)}(v, v')$ to denote the set of paths in $g(s)$ from v to v':

$$\Pi(v, v') \stackrel{\text{def}}{=} \{\pi \in \Pi \mid src(\pi) = v, dst(\pi) = v'\}$$

3.2 Heap Primitives Provided by PHALANX

The primitives supported in PHALANX and their semantics are shown in Table 1. The primitive `dom` which returns `boolean`. All other the primitives return `JMLObjectSet`, a set implementing standard set-operations such as membership test, intersection, and size.

We use a standard Java semantics to evaluate the formal parameters of the primitives and require that the their values are non-null. The semantics of a parameter `o` of type `Object` in a state s is the node in $g(s)$ that represent the object pointed to by `o` in state s. Similarly, the semantics of a parameter `avoid` of type array of `Object`, is the set of nodes in $g(s)$ that represent the objects in the array. We abuse the notations slightly and use `o` and `a` to represent both the parameters and their semantics in s.

EXAMPLE 3.2. *Consider the state of the JdbF program of Fig. 1 shown in Fig. 2. The following JML assertion uses the primitive* `pred` *to check the* `Connection` *object pointed to by c has exactly one predecessor:* `pred(c).size() == 1`*. This assertion is satisfied because every* `Connection` *object has only the corresponding* `ConnectionSource` *object as its predecessor from the heap.*

The assertion `dom(stack(Thread.currentThread()), c)` *is not satisfied because the* `Connection` *object in the center is pointed to from the stacks of both client threads. Suppose that the client threads are correctly synchronized, i.e., only one of them has a stack pointer to c. Then, according to the semantics defined above, the assertion holds.*

Note that when the query `dom(o1,o2)` is evaluated, the thread that invokes it is the current thread ct, and it has at least one stack pointer to the object pointed to by $o2$, namely the argument $o2$ of the query. Therefore, the object pointed to by $o2$ is not dominated by the object pointed to by $o1$. Intuitively, the stack pointer to the object pointed to by $o2$ should be ignored for the purpose of domination check, because it exists solely for invoking the query, but does not violate the domination/ownership relation that the user intended to check with this query. Similarly, it is possible that there are multiple pointers from other stack frames of the current thread that are there solely for acquiring the pointer to $o2$ to invoke the query.

To address it, the semantics of `dom` ignores all paths that go through the stack of the current thread. It is possible that this semantics is ignoring paths that violate thread-local ownership through the stack. Note that the semantics does consider paths that go through stacks of other running threads as violating the domination.

3.3 Common Heap Queries and their Usage Scenarios

Table 2 shows how to express common heap queries using JML assertions with the heap primitives provided by PHALANX. The last column lists names of heap probes — PHALANX runtime methods that efficiently implement these queries, as described in Section 4.

We illustrate the use of these queries with a series of examples drawn from common programming situations and design patterns, showing the effectiveness of heap queries for debugging the code. [1]

Note that queries related to threads (e.g., getThreadReach and isThreadOwned) can have 3 versions: stack only, thread object only, and both. The semantics supports such distinctions by having 3 nodes for every running thread of $g(s)$, where the nodes labelled with stack and running do not represent objects, but were introduced to refine the notion of reachability from threads. It is useful to have a shorthand which returns all the "roots" of a given thread; we provide it along with the PHALANX primitives. `roots(Thread t)` returns the JMLObjectSet which consists of all objects pointed from the thread's stack and the thread object itself:

$$\texttt{roots(Thread t)} \stackrel{\text{def}}{=} \texttt{stack(t).insert(t)}$$

Confinement A standard pattern for maintaining sanity in concurrent programming is *object confinement* [20]: all references to a resource o come from a single thread t. This pattern guarantees that only one thread at a time can modify o, and thus no synchronization on o is necessary. The heap assertion `dom(t,r)` allows t to confirm that r is confined.

Fig. 3 shows a code fragment adapted from `SimpleWebServer` [18] with an additional heap assertion. In SimpleWebServer, a new thread is created for every request received by the web-server. The new `RequestThread` is passed a `Socket` through which it communicates with the client. The heap assertion in the method `run` of `RequestThread` checks that the `Socket` is confined to its `RequestThread`.

Escape Analysis It is often desirable to check that an object does not escape from a procedure. This is particularly important in a concurrent setting where exposing a heap reference to an object without proper synchronization might lead to an undesirable modification by another thread. Using

[1] The source code of our examples available from http://heapassertions.unfuddle.com/projects/33697/repositories, login and password are oopsla09. Due to restrictions on distributing the production VM, we cannot make it available at this time.

Name	Semantics	Description
`running()`	$\{r.obj \mid L(r) = \textit{running}\}$	Running threads
`stack(Thread t)`	$\{v \mid (r.stack, v) \in E \wedge L(r) = \textit{running} \wedge r.obj = \texttt{t}\}$	Objects pointed-to from the stack of thread `t`
`reach()`	$\{v \mid L(v) \in \textit{Classes} \wedge \Pi(\textit{root}, v) \neq \emptyset\}$	Reachable objects
`reach(Object o)`	$\{v \mid \Pi(\texttt{o}, v) \neq \emptyset\}$	Objects reachable from object `o`
`reach(Object o,Object[] a)`	$\{v \mid \exists \pi \in \Pi(\texttt{o}, v').\textit{nodes}(\pi) \cap \texttt{a} = \emptyset\}$	Objects reachable from object `o` without going through any of the objects in `a`
`pred(Object o)`	$\{v \mid (v, \texttt{o}) \in E \wedge L(v) \in \textit{Classes} \wedge \Pi(\textit{root}, v) \neq \emptyset\}$	Reachable objects pointing to object `o`
`dom(Object o1,Object o2)`	$\exists \pi \in \Pi(\texttt{o1}, \texttt{o2}) \wedge$ $\forall \pi \in \Pi(\textit{root}, \texttt{o2})\texttt{o1} \in \textit{nodes}(\pi) \vee \textit{ct}.\textit{stack} \in \textit{nodes}(\pi)$	There is a path from `o1` to `o2`, and every path from root to `o2` go through `o1`

Table 1. Primitives for reasoning about the heap and their semantics in state s, where $g(s) = \langle V, E, L \rangle$.

Query	Description	Probe Name
`pred(o).size() > 0`	Is *o* pointed to by a heap object?	isHeap(Object o)
`pred(o).size() > 1`	Is *o* pointed to by two or more heap objects?	isShared(Object o)
`reach(src).has(dst)`	Is *dst* reachable from *src*?	isReachable(Object src, Object dst)
`!(exists Object v; reach(o1).has(v) ; reach(o2).has(v))`	Is there an object reachable from both o_1 and o_2?	isDisjoint(Object o_1, Object o_2)
`!(exists Object v ; reach(o).has(v) ; !dom(o,v))`	Does *o* dominate all objects reachable from it?	isUniqueOwner(Object *o*)
`!reach(o1,cut).has(o2)`	Does every path from o_1 to o_2 go through an object in *cut*	reachThrough(Object o_1, o_2, Object[] cut)
`dom(Thread.currentThread(), o)`	Does the current thread dominate *o*?	isObjectOwned(Object o_1, Object o_2)
`dom(stack(Thread.currentThread()), o)`	Does the current thread's stack dominate *o*?	isThreadStackOwned(Object *o*)
`dom(roots(Thread.currentThread()), o)`	Does the current thread dominate *o*?	isThreadOwned(Object *o*)
`{Thread t | running().has(t) && (reach(t,avoid).has(o) || reach(stack(t),avoid).has(o))}`	Threads from which object *o* is reachable not through *avoid*	getThreadReach(Object o, Object [] avoid)

Table 2. Common heap queries and their corresponding probe names.

the primitive `pred(o)`, the programmer can check whether or not there are heap references to `o`.

The `run()` method of `SimpleWebServer` class in Fig. 3 has a simple defense against Denial of Service (DoS) attacks. The server keeps a set `clients` of currently connected clients. When a new client connects, it is added to the set. Multiple connections from the same client are scrutinized by the `suspicious` method, checking for evidence that the client may be executing a DoS attack on the server, and may be rejected. The decision must be made quickly, to keep the defense from being a denial of service in its own right.

In particular we do not wish to synchronize. This is legitimate if `clients` is confined to `run()`. But, `clients` is passed to `suspicious`, so confinement is not instantly obvious. We can check (during testing rather than deployment) that `clients` is only used by the server thread, and no references to it are stored in the heap, by the heap assertion `pred(clients).size() == 0`.

Wrappers The Wrapper design pattern is a staple of programming. When one is trying to glue two big systems A and B together, A's objects will sometimes need to refer to B's, and vice-versa. It is generally impractical to fully unify two independently-developed object hierarchies. So the programmers create *wrapper classes*, WrapBObject, which present an A-style interface for a B object.

It is generally desirable that all access to B objects from A go through the wrappers. (Not all access in the system: B will generally access its own objects directly.) PHALANX provides a straightforward check for this: `!Phalanx.reach(a, new Object[]{w}).has(w.b)` returns true iff all paths from `a` to `w.b` go through the wrapper `w`. The middle argument can be an array, in case there are several legitimate access paths.

Splitting tasks among threads A common way to speed up a computation on a multicore machine is to have multiple threads working on disjoint, non-interfering parts of the same problem. Examples include implementation of matrix multiplications, union-find, mergesort, and tree traversals. The correctness of the computation often relies on the data processed by one thread being disjoint from data processed by all others. This property might not be obvious when the data is stored in multiple collections. The programmer can use the heap query `isDisjoint` to check this assumption.

```
public class SimpleWebServer ... {
  public void run() {
   Set clients = new HashSet();
   while (_running) {
    Socket wsocket = _serverSocket.accept();
    InetAddress address = wsocket.getInetAddress();
    //@ assert (Phalanx.pred(clients).size() == 0);
    if (clients.contains(address))
      if (suspicious(clients,address)) wsocket.close();
    else {
     clients.add(address);
     RequestThread rt = new RequestThread(wsocket,_rootDir);
     wsocket = null;
     rt.start();
    }
   }
 }
 // @return true when DoS attack from address is suspected.
 private Boolean suspicious(Set clients,InetAddress address){
  ...
 }
 ...
}
public class RequestThread extends Thread {
 private Socket _socket;
 ...
 public RequestThread (Socket socket, File rootDir) {
  _socket = socket;
  _rootDir = dir;
 }
 public void run() {
   // @assert Phalanx.dom(this,_socket);
   ...
 }
}
```

Figure 3. Confinement of `Socket` in `RequestThread`.

4. Heap Probes Algorithms

In this section, we present the design of our new parallel algorithms for evaluation of heap queries. We also discuss some high-level challenges we faced when implementing and integrating these algorithms into a production-grade virtual machine. Additional lower-level implementation challenges are discussed in Section 5.

Rationale In this work, we are interested in: (i) answering expressive heap queries, including queries that involve path properties; (ii) keeping the auxiliary space required to answer a query to a minimum.

These two requirements preclude approaches based on pre-computation of a transitive closure graph. Transitive closure can be pre-computed in $O(V*(V+E))$ operations (for a heap with V objects and E references), and updated using incremental algorithms (*e.g.*, Roditty [32]). However, pre-computation incurs a prohibitive space overhead, as well as a significant time overhead when the heap is modified frequently. Furthermore, this approach does not support path queries which we show to be of practical importance (e.g., for ownership and reachability with an avoidance set).

For these reasons, we chose to evaluate queries directly on the heap graph. The worst-case time complexity for most our algorithms is $O(V + E)$ operations for a heap with V objects and E references. Evaluating P of them could take $O(P(V + E))$ operations. However, in practice, we expect $P << V$ and heap queries are much less frequent

```
trace(t_m)
 while (t_m.pending ≠ ∅)
  remove s from t_m.pending
  for each o ∈ {v | (s,v) ∈ E}
   trace-step(s, o)
   mark-object(t_m, o)

tag-object(t_m, o)
 if (tag-step(o) = false)
  return false
 atomic
  if (o ∉ Marked)
   Marked ← Marked ∪ {o};
   return true
  else return false

mark-object(t_m, o)
 if (tag-object(t_m, o) = true)
  push-object(t_m, o)

push-object(t_m, o)
 t_m.pending ← t_m.pending ∪ {o}

mark-thread(t_m, t_a)
 for each o ∈ roots(t_a.stack)
  mark-object(t_m, o)
 mark-object(t_m, t_a.obj)

mark-roots(t_m, T)
 for each t_a ∈ T
  mark-thread(t_m, t_a)
 mark-object(t_m, static)
```

Figure 4. Basic Components

than heap updates, and hence for large program heaps, this approach is likely to be superior to incrementally updating a pre-computed transitive closure when the heap is modified.

To speed up the evaluation of heap queries, we designed new parallel algorithms that can leverage all of the available cores in the system for evaluating a query. Our algorithms operate by stopping the application, evaluating the heap query in parallel on the program heap, and then resuming the application.

We have implemented our parallel algorithms as part of a production virtual machine. The VM already contains much of the necessary infrastructure required for efficient implementation of these algorithms. However, production grade virtual machines are complex pieces of code, and correctly implementing the algorithms is quite challenging.

Heap Query Evaluation vs. Garbage Collection Many of our heap queries check path properties and therefore *cannot be implemented by piggybacking a tracing garbage collector*. There are many variants of tracing collectors (e.g. what regions of the heap it focuses on), when it is triggered (e.g work-based, time-based, etc) and what actions are taken when it encounters an object (e.g. copying or marking). Regardless, at its core, a tracing collector computes reachability.

However, heap probes such as *isObjectOwned* compute path properties, and cannot be evaluated by computing transitive reachability.

Hence, rather than trying to piggyback on an existing collector (e.g. as in [1]), we focus on a more flexible solution: using existing components of the runtime to efficiently implement our parallel algorithms for evaluating heap queries. Specifically, we reuse components used by a parallel garbage collector to compute transitive reachability, but put them together to yield the more general computations required by our queries.

Performing Garbage Collection during Heap Query Evaluation Our parallel algorithms are designed and integrated

into the virtual machine such that the work done during query evaluation can be re-used by the garbage collector. Hence, in addition to answering the required heap query, we have the option to perform garbage collection right after the probe finishes, *leveraging the tracing work done by the probe*. That is, the collector can piggyback on our heap query evaluation, rather than opposite, which is not possible for most of our algorithms.

4.1 Virtual Machine Components Used By Parallel Probe Implementations

Much of the machinery necessary to implement our parallel algorithms is already present in the virtual machine and is used by the existing parallel garbage collector in a specific manner. We now describe the machinery that is used by our implementation.

When the virtual machine starts, a set of *evaluator threads* T_m are created, one for each core. These evaluator threads are initially blocked. The runtime uses these threads for parallel garbage collection. We use these threads for evaluation of our heap queries.

Heap Traversal Components The basic components of a parallel garbage collector are shown in Fig. 4. Each procedure has an evaluator thread t_m as a parameter. The set T_a denotes the set of (running) application threads in the system at the time a probe is invoked. Thread-local variables are written with the prefix of the thread to avoid confusion: e.g., $t_m.temp$ is a local variable $temp$ of thread t_m. For an application thread t_a, we use $t_a.stack$ and $t_a.obj$ to denote the thread's stack and the thread object. In pseudocode we assume that all sets are initially empty.

The procedure `mark-thread()` marks the objects directly pointed to by an application (running) thread stack and thread-object, but does not perform further traversal from that set. The procedure `trace()` performs heap traversal to computes the set of objects reachable from the set $t_m.pending$. The marking proceeds as usual in garbage collection, but we have added callback procedures `trace-step` and `tag-step`, which are called on each newly-encountered reference. Different implementations of the various heap probes customize these routines in specific ways. The default return value of `tag-step` is $true$.

Synchronization Primitives Our parallel algorithms require careful attention to synchronization. Some probes are implemented using a multi-phase traversal, and thus need synchronization barriers. Our implementation uses two main forms of synchronization, as follows:

Barrier: A standard barrier is provided by the function `barrier()`. When an evaluator thread calls `barrier()`, it blocks and waits for all other evaluator threads to arrive at the barrier. When they have all called `barrier()`, all of the evaluator threads are released to continue.

Barrier with Master thread: On startup of the virtual machine, one of the evaluator threads is designated as the *master* thread. When a thread calls the function `barrier-and--release-master()`, it blocks, just like in the case of `barrier()`. When all of the evaluator threads have called the function, only the master thread is released and allowed to continue, while the other threads remain blocked. These threads remain blocked until the master releases them by a call to the function `release-blocked-evaluators()`. The procedure `barrier-and-release-master()` returns true for the master thread and false for all others threads.

Object Sets The virtual machine already provides efficient implementation of various set operations. This facility is typically used by the garbage collector to put objects in a marked set (implemented via marked bits that can be efficiently set and cleared). For our probes, when necessary, we use this capability to create and manipulate other sets (such as the $Owned$ set).

4.2 Parallel Implementation of Individual Probes

Next, we present the parallel algorithms used for evaluating heap probes. Our algorithms are not specific to a particular runtime, and only rely on the common primitives discussed in Section 4.1.

4.2.1 Single-Phase Algorithms

We begin by describing three probes that only require a single phase during their evaluation. In Section 4.2.2, we present more involved algorithms for handling path properties such as object ownership, domination, as well as algorithm checking disjointness.

isReachable Fig. 5(a) shows the algorithm for evaluating our simplest probe, `isReachable(source, target)`. This probes operates by marking the set of objects reachable from `source`, and therefore resembles tracing garbage collection. Note that the code shown in the figure is executed in parallel by a all evaluation threads. The probe starts by marking the `source` object, and then traces from it. All evaluation threads eventually block at `barrier-and-release-master()`. When this happens, the object pointed to by `target` is guaranteed to be marked if it is reachable from `source`. At this point, the master thread sets the return value based on whether the `target` object is marked (this need only be done by one thread, hence the use of the master) and then releases the other evaluator threads. `trace-step()` is not needed in this probe and is left empty (hence omitted from figure).

isShared Fig. 5(b) shows the implementation of the probe `isShared(o)`. In this algorithm, every evaluator thread uses a private set $t_m.sources$ to record the objects pointing to o that it encountered during its traversal. By using private sets, we avoid the need for synchronization between evaluator threads during the tracing phase (e.g., this is an alternative to incrementing a shared counter). When the tracing phase completes, evaluator threads combine their local sets into a global view by updating a global set $allsources$ under a

```
isReachable(t_m, source, target)
  mark-object(t_m, source)
  trace(t_m)
  if barrier-and-release-master()
    if (target ∈ Marked) result ← true
    else result ← false
    release-blocked-evaluators()
```

(a) Reachability

```
isShared(t_m, o)
  t_m.sources ← ∅
  mark-threads(t_m, T_a)
  trace(t_m)
  lock(allsources)
  allsources ← allsources ∪ t_m.sources
  unlock(allsources)
  if barrier-and-release-master()
    if |allsources| > 1 result ← true
    else result ← false
    release-blocked-evaluators()

trace-step(s, t)
  if (o = t) t_m.sources ← t_m.sources ∪ {s}
```

(b) Shared from Heap

```
isThreadOwned(t_m, t_a, o)
  mark-roots(t_m, T_a \ {t_a})
  trace(t_m)
  if barrier-and-release-master()
    if (o ∈ Marked) result ← false
    else result ← true
    release-blocked-evaluators()
```

(c) Thread ownership

Figure 5. Single-phase parallel algorithms for checking reachability, sharing, and thread ownership.

lock. Combining the local sets is required as it is possible that o is shared but each parallel evaluator t_m reached it only once during its traversal. Finally, the threads synchronize, the master thread computes the result (if we have more than one object in $allsource$, we return *true*, otherwise *false*) and releases the rest of the evaluator threads.

For clarity of presentation we have omitted some trivial implementation details from the figures such as sizes of $t_m.sources$ and $allsources$. During tracing, each evaluator thread stops recording once $t_m.sources$ contains more than two objects, as the object o is then known to be shared. In this case, the probe can return *true* immediately if we do not require garbage collection to start after probe computation.

isThreadOwned The implementation of the probe `isThreadOwned(`t_a`, o)` is shown in Fig. 5(c). This probe checks if object o is reachable *only* from the calling application thread t_a. To compute this, we trace from all application threads except from t_a. If object o is marked, then it is not owned by t_a and we return *false*, otherwise we return *true*. Note that this probe assumes that t_a is the application thread that invokes the probe, hence object o is always reachable from t_a (and we do not need to explicitly check that).

The operation of this probe is quite similar to tracing collectors, with the key difference being that there is specific order on the way threads are processed. Note that if we would like to perform garbage collection during this probe, after the probe completes, we can proceed to mark and trace from only the roots of the current application thread t_a. That is, collection can reuse the rest of the work that was done for the probe.

4.2.2 Multi-Phase Algorithms

We now describe more complicated algorithms that requires multiple synchronization barriers during their operation.

isObjectOwned The implementation of the probe `isObjectOwned(source, target)` is shown in Fig. 6(a). Recall that this probe only returns *true* when all *heap paths* to $target$ go through $source$ and there is at least one such path. The algorithm uses a special sequence for processing nodes, and only uses the single set $Marked$. The basic idea is to mark $source$ without tracing from it, and then trace through all other roots. Since source is marked during tracing, it won't be traced through and all objects that are reachable only through source will remain unmarked. The actual algorithm is a bit more involved, and its schematic operation is shown in Fig. 7. In the figure, barriers between phases are shown as vertical lines labelled with the name of the updated phase. The main stages of the algorithm are:

- mark $source$ — the algorithm uses `tag-object()` to mark the $source$ object *without tracing from it*.
- mark all objects pointed to from the roots (except $target$) without tracing from them yet. The purpose this phase is to avoid marking $target$ if it is pointed directly from the roots as we want to reason only about heap paths.
- perform tracing — during tracing, if evaluation threads encounter the object $source$, they do not trace from it because it is already in the $Marked$ set. Upon completion of the tracing phase, threads synchronize and check whether $target$ is marked. If it is marked, then the probe returns *false*. Otherwise we proceed to check if $target$ is reachable from $source$ and if it is, the result is set to *true*.

Finally, note that we need to manage the object $source$ carefully, because it has been marked already but was not placed in the $pending$ set. That is, if $target$ is not marked and we requested garbage collection to be performed during this probe, we need to make sure that $source$ is added to $pending$ before we continue with the collection operation.

One of the inherent challenges of implementing probes of this type in a language runtime, is dealing with stack pointers. In particular, objects $source$ and $target$ are always reachable from the stack of the application thread that invoked the heap probe. Our current implementation focuses on domination through heap paths, and ignores all stack pointers to $target$. Alternative implementations could identify which stack pointers to consider and which stack point-

```
isObjectOwned(t_m, source, target)
  tag-object(t_m, source)
  result ← false
  phase ← skip
  barrier()
  mark-roots(t_m, T_a)
  barrier()
  phase ← none
  trace(t_m)
  barrier()
  if (target ∉ Marked)
    barrier()
    push-object(t_m, source)
    trace(t_m)
    if barrier-and-release-master()
      if (target ∈ Marked) result ← true
      release-blocked-evaluators()

tag-step(t)
  if (phase = skip ∧ t = target)
    return false
```

(a) Object Ownership

```
thread[] getThreadReach(t_m, o, avoid[])
  foreach t_a ∈ T_a
    foreach a ∈ avoid[]
      tag-object(t_m, a)
    barrier()
    mark-thread(t_m, t_a)
    trace(t_m)
    if barrier-and-release-master()
      if (o ∈ Marked) out ← out ∪ t_a.obj
      Marked ← ∅
      release-blocked-evaluators()
  return out
```

(b) Get Reaching Threads

```
reachThrough(t_m, o1, cut[], o2)
  result ← true
  foreach c ∈ cut[]
    tag-object(t_m, c)
  barrier()
  mark-object(t_m, o1)
  trace(t_m)
  barrier()
  if (o2 ∈ Marked) result ← false
```

(c) Dominates Through

Figure 6. Parallel algorithms for path properties (ownership, reachability with avoidance set)

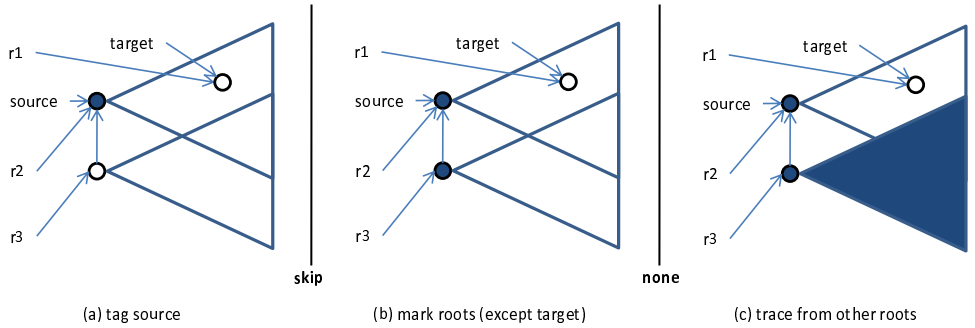

Figure 7. Schematic operation of `isObjectOwned(source, target)`. $r1, r2, r3$ and $source$,$target$ are roots. Circle nodes denote heap allocated objects, triangles denote parts of the heap that are transitively reachable from the object they emanate from. Vertical lines denote synchronization barriers used in the algorithm. In this example, $target$ is owned by $source$.

ers to ignore, but this is very challenging in practice, especially due to various JIT optimizations.

getThreadReach The implementation of the probe `getThreadReach(o, avoid)` is shown in Fig. 6(b). This probe returns all application thread objects which can reach object o without going through any object in the $avoid$ set. We consider each application thread t_a in turn, to see if o can be reached from that thread. As in `isThreadOwned`, we first tag all objects in $avoid$ set. Then we compute the transitive closure from that thread. If after that, o is marked, then the application thread is inserted into the out set, otherwise, we do not insert it. Note that, after processing each application thread, the $Marked$ set is initialized to $\emptyset$. Practically, this is possible in our virtual machine because the marked bits for each object can reside in a continuous memory region outside of the object space, making it easy to re-initialize that space. The set out is assumed to be initialized to $\emptyset$ on startup of the probe. Also note that this probe tracks reachability from both thread stacks and thread objects. We can specialize it further as described in Table 2 to track reachability only from thread stacks or only from thread objects.

reachThrough The implementation of the probe `reachThrough(o1, cut, o2)` is shown in Fig. 6(c). This probe checks that all paths from object $o1$ to object $o2$ go through at least one object in the set cut. The algorithm uses a similar trick to `isObjectOwned`. First, it marks all the objects in the set cut but does not trace from them. Then it marks and traces from object $o1$. If during this process, we encounter an object in the cut, we will not trace through the object as it was already marked initially. At the end of the tracing from $o1$, if we see that object $o2$ is marked, then there must have been a path from $o1$ to $o2$ not going through

```
isDisjoint(t_m, o1, o2)
 result ← true
 phase ← dual
 mark-object(t_m, o1)
 trace(t_m)
 if (o2 ∈ Marked) result ← false
 barrier()
 phase ← check
 t_m.temp ← result
 barrier()
 if (t_m.temp = true)
   mark-object(t_m, o2)
   trace(t_m)
   barrier()
 phase ← none

trace-step(s, t)
 if (phase = dual)  Owned ← Owned ∪ {t}
 else if (phase = check ∧ t ∈ Owned)
   result ← false
```

Figure 8. Parallel algorithm for disjointness.

any object in the set cut. In that case, the probe returns *false*. Otherwise, the probe returns *true*.

isDisjoint Fig. 8 shows the implementation of the probe `isDisjoint(o1,o2)`. Note that this property cannot be computed by a single reachability computation. In particular, the computation of this probe *requires two sets of objects* (as opposed to the single $Marked$ set when computing reachability). The basic idea is to compute the set of objects reachable from $o1$ and intersect it with the set of objects reachable from $o2$. However, we do this more efficiently, and in a way that guarantees that work performed by probe computation can be re-used for garbage collection.

- Initialization — each thread sets the shared $result$ variable to *true* and sets the phase to double marking ($dual$).
- The $dual$ phase (tracing from $o1$) — during this phase, traced objects are added to the $Owned$ set and to the $Marked$ set. The set $Owned$ is not updated in later phases, and identifies the objects reachable from $o1$. When a thread finishes tracing, it checks whether $o2$ is marked and if it is, sets $result$ to *false*.
- switching the phase — all threads synchronizes via the barrier to ensure completion of the $dual$ phase. All threads then synchronously switch the phase to $check$ and read the value of $result$. If after the barrier the result is still *true*, then all threads attempt to trace from $o2$.
- The $check$ phase (tracing from $o2$) — in this phase, each evaluator thread checks if it encounters an object in the $Owned$ set during tracing, and sets the result to *false* if it does. Upon completion, we synchronize the evaluator threads and switch the final phase ($none$).
- final phase — after the first evaluator thread changes the phase to $none$, all objects reachable from $o1$ and $o2$ are guaranteed to be in the $Marked$ set. This means that if we would like to proceed with garbage collection, we can do so in the usual manner.

The probe `isUniqueOwner` is implemented by an elaboration of this scheme and we do not present it here.

5. Implementation Details

5.1 Modifying the JML Compiler

We modified the JML compiler to identify common queries and translate them into calls to heap probes in the PHALANX runtime. Our replacement uses simple pattern matching on the AST, and is performed at the code generation phase of the JML compiler.

Specifically, we modify the translation of quantified expressions. The original compiler does not produce any code for quantified expressions, as their evaluation cost may be prohibitive in practice (the compiler notifies the user that no code is generated in these cases). We modify the compiler to generate the appropriate calls to the PHALANX runtime when the quantified expression match one of the common heap queries.

Technically, we managed to keep our changes to minimum by modifying the translation step in the compiler, where java code is generated from JML expressions. In this phase, we replace the quantified expression by a method call expression invoking the appropriate probe method in the PHALANX runtime. We extract the arguments for the probe from the original JML expression, and use it to construct a new valid expression for translation.

5.2 Modifying the VM

Heap probes are implemented on the QVM platform [4], which is based on IBM's J9 production virtual machine.

Intercepting calls to Heap Queries User-level code interacts with our heap probes via specially provided library. All of the probes in this library call a single designated internal method, which is intercepted by the VM on JIT compilation. If intercepted, we patch that internal library call with a call to the internal VM functions required for running the probes.

Sharing Components between Heap Probes and Garbage Collection Some of the components used by our heap probes are also used by garbage collection. Care must be taken to make sure that changes to these components do not affect the operation of the normal collector. For example, as mentioned in Section 4.1, heap traversal routines now contain calls to `trace-step` or `tag-step`, which should not be invoked during normal garbage collection cycles.

To distinguish an evaluator thread performing heap query evaluation from a thread performing garbage collection, we re-use some of the free space in each evaluator-thread structure to denote its kind. The kind of a thread is set when the operation starts (i.e., query evaluation or garbage collection) and is used only when necessary. An alternative implementation strategy would have been to add arguments to exist-

ing functions, but such changes would have spanned a large number of VM modules, making implementation harder and more error-prone.

In general, some probes make heavy use of the free space available in the evaluator-thread structure. For example, rather than storing information in a shared location that requires synchronization on each access, some of the algorithms store the information locally in the evaluator-thread structure, and synchronize and merge it into a shared result only at the end of query computation.

Interaction between Heap Probes and Garbage Collection Some of our probes require marking an object without tracing from it (e.g., $isObjectOwned$). To enable reuse of the work performed by heap probe for query evaluation for garbage collection, we must keep track of such objects. If GC is requested when the probe computation finishes, these tracked objects must be pushed on the marking stack for garbage collection. Failure to do so might lead to sweeping of live objects (the ones reachable from the tracked objects).

If the probe is not required to perform garbage collection and returns immediately, we need to make sure that all intermediate state used by the probe that may be used by the normal collection cycle is reset (e.g. the $Marked$ and the $pending$ sets).

Load Balancing for Heap Probes The main challenge we addressed when designing the parallel algorithms is correct placement of synchronization barriers. Our parallel algorithms for heap probe evaluation can utilize any of the existing parallel algorithms for heap traversal, with any kind of load balancing (e.g. work dealing, work stealing), and can benefit from developments in load balancing techniques (e.g. [24]).

In general, as with all dynamic load balancing schemes, it is always possible to come up with a demonic graph topology (e.g. a singly linked list) that defeats parallelism of our probe algorithms. However, with increasing heap sizes and the increasing variety of structures found in the heap, it is likely that the parallel heap traversal algorithms will perform well in most applications.

6. Experimental Evaluation

In this section, we describe the evaluation of our implementation. We evaluate the implementation in several respects:

- evaluate the PHALANX implementation compared to a reference implementation in JVMTI
- evaluate the scalability of parallel query evaluation
- evaluate the usefulness of the queries in real applications.

6.1 Implementation of Heap Probes using JVMTI

The Java Virtual Machine Tools Interface (JVMTI) is a virtual machine independent powerful native programming interface for use by tools that need access to JVM state for profiling, debugging, monitoring, and coverage analysis. It provides a way to inspect the state and to control the execution of applications running in the Java virtual machine.

Recently, the JVMTI was extended with new methods, enabling the programmer to write to request traversal of the entire heap. This new capability allows us to implement our algorithms using JVMTI, as an alternative to the QVM-based implementation presented earlier.

QVM vs. JVMTI The key advantage of implementing heap probes using JVMTI is its portability across different virtual machines. However, heap traversal in JVMTI is required to "stop-the-world": explicitly stop all threads except the thread that invokes the traversal. That is, if several threads invoke heap traversal of JVMTI, their traversals will be performed sequentially. Certainly, if parallel traversal becomes available in future versions of JVMTI, our parallel algorithms would be applicable.

We implemented sequential simplified variants of all of the probes in JVMTI. We could have used the parallel algorithms as is and just run them sequentially, but we wanted to avoid unnecessary synchronization when using the algorithms in a pure sequential setting.

We evaluated the performance of our heap probes implemented using JVMTI in several different virtual machines, from different vendors, including Sun and IBM. Implementation using JVMTI is about five (5) times slower on average than the QVM implementation on a single CPU (see Fig. 10). QVM-based implementation re-uses existing machinery inside the VM, tailored for heap traversals.

6.2 Benefits of Parallelization

We performed experiments with synthetic benchmarks with large heap size, complex heap structure, and different heap probes, to evaluate the scalability of the parallel algorithms with respect to increasing number of worker threads. We ran our experiments on 8-core 2.4 GHz AMD Opteron. Fig. 9 shows almost linear speedup in the evaluation time. In this benchmark, we evaluate the probe `isShared` on a heap with numerous threads, arrays and linked objects, total of nearly 8.3 million objects. To stress test the parallel implementation, we designed the benchmark such that the evaluation of the probe requires traversing the entire heap (i.e., the probe `isShared` returns false).

We also evaluate the parallel algorithms on heaps of different sizes. Fig. 11 shows that the speedup gained by increasing the number of cores is uniform for heaps of different sizes. Finally, Fig. 10 shows that as the heap size increases, the evaluation time using JVMTI implementation is higher (i.e., slower) than the time it takes the QVM implementation even on a single core. Moreover, the difference between evaluation time on a single core and on 8 cores increases with heap size. To summarize, this benchmark shows significant benefit of parallelization and shows that the benefit increases with heap size.

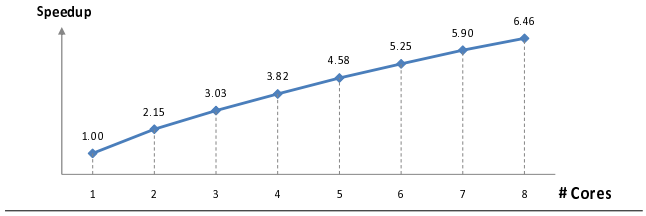

Figure 9. Speedup in probe evaluation time with increasing number of cores.

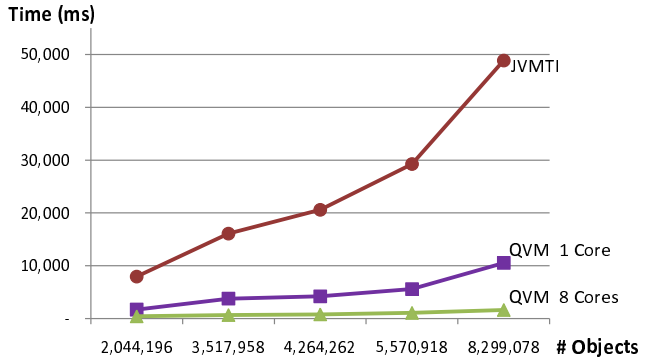

Figure 10. Probe evaluation time for increasing heap sizes.

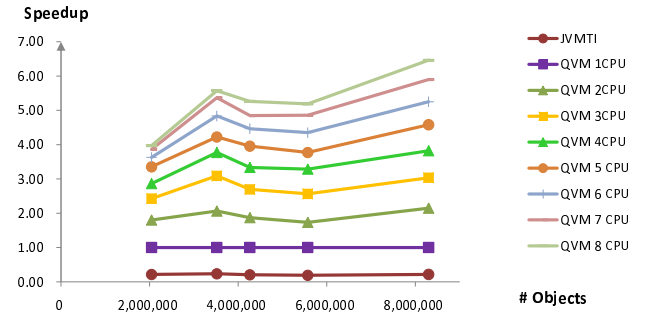

Figure 11. Probe evaluation time for increasing heap sizes and increasing number of cores.

6.3 Applications

Adding meaningful heap assertions to real world applications requires deep understanding of program invariants, which may be difficult even for the original developer of the code.

Adding arbitrary assertions just to measure performance misses the mark, because in many of the applications we considered the assertions might be evaluated on different heap graphs during different executions. Further, the cost of evaluating a heap query depends on its answer, for examples, when an object is not shared, evaluating `isShared` on it requires traversing the entire heap. This may lead to significant variance in the cost of assertions, especially in real applications where the behavior may be non-deterministic and depend on an elaborate environment (e.g., network connections).

To eliminate these factors, we conducted a controlled evaluation of performance with the synthetic benchmarks mentioned earlier.

In this section, we evaluate the usefulness of heap assertions on several real-world applications in two ways:

- we pick two applications that we are fairly familiar with, and manually add meaningful heap assertions after careful inspection of the code.
- we use a script to add a large number of assertions based on two common scenarios that are of general applicability.

Application	LOC	Probes	Violations
AOI	111,333	10	0
Azureus	425,367	334	16
Freemind	70,483	16	2
Frostwire	245,959	184	2
JEdit	93,790	66	0
jrisk	20,807	45	12
rssowl	74,280	95	3
tvbrowser	105,471	40	1
TVLA	57,594	10	0

Table 3. Evaluation in real-world applications

Table 3 shows the applications we use in this study. For each application, the table shows the number of lines of Java code (generated using David A. Wheeler's SLOCCount), and the number of probes we inserted into the code.

Manually Added Assertions

In `TVLA` and `AoI` we manually added assertions that were picked after careful examination of the code.

AoI ArtOfIllusion (AoI) is an open source application for 3-D modelling, animation and rendering, written entirely in Java. Our benchmark consists of loading and rendering 11 existing 3-D models, which present complex scenes with many hundreds of 3-D objects, as well as several lights and cameras, created by professional artists for production purposes.

The 3-D objects in a scene are arranged hierarchically, such that moving, scaling and rotating a parent object can result in the children objects also being transformed. Moreover, several 3-D objects can share graphic elements, such as textures and skeleton objects for controlling animation.

To speed up rendering, AoI automatically creates a number of worker threads based on the number of available processors. Each worker thread repeatedly executes small tasks such as tracing a ray through a single pixel or shading a triangle.

We added assertions to check (i) structural properties of the scene and (ii) correct coordination of rendering threads.

In this application, the assertions we added were not violated in all of our test runs, and incurred no observable slowdown in the operation of the application when running with PHALANX.

TVLA TVLA [22] is a parametric framework for shape analysis that can be easily instantiated to create different kinds of analyzers for checking properties of programs that use linked data structures. In TVLA, program states are represented as three-valued logical structures, and the programs transition system is defined using first-order logical formulas.

To reduce its space usage, TVLA uses shared representation of logical structures and formulas. We check that the implementation of abstract transformers copies the structures it modifies. We also added assertions to check that operations of formulas do not violate the structural properties of formulas. Another optimization, in the chaotic iteration of TVLA, maintains a pending list of structures to process only the structures that changed. We added assertion to check that the pending list is correctly manipulated.

In this application, the assertions we added were not violated in all of our test runs.

Heap Assertions Added Automatically

For the rest of the applications, we used scripts to automatically insert heap-assertions for several scenarios. Unfortunately, we could not make the JML compiler compile these entire real-world applications, and therefore in these applications we added direct calls the PHALANX probes.

Most of the applications we considered are interactive ones, and the specific probes being executed therefore often depend on user interactions.

Sharing of Disposed Resources Many of the applications in Table 3 use a GUI based on the Standard Widget Toolkit (SWT) [14]. GUI elements in SWT have to be manually allocated and disposed by the programmer. Disposing an SWT resource is performed by explicitly invoking the method `dispose()` on it. Failure to properly dispose SWT resources leads to leakage of OS-level resources and may gradually hinder performance and even lead to a system crash. In many cases, programming patterns help. But widely-shared resources like `Color`s, `Font`s, and `Image`s, are notoriously hard to manage properly.

Our heap assertions allow the programmer to check, at the point of calling `dispose()`, whether the resource about to be disposed is shared. Using our heap queries, the programmer can also get a list of threads that can reach the resource, which is extremely useful for debugging.

To identify such potential cases, we replace code of the form:

```
exp.dispose();
```

with code of the form

```
if (Phalanx.isShared(exp))
 Phalanx.warning("disposal of shared resource" + exp);
exp.dispose();
}
```

We ran our benchmarks with the added assertions, for the most part, the frequency of resource disposal is low enough such that the application exhibits very little observable slowdown when running on PHALANX. We observed that in several of the applications our assertions are sometimes violated, and disposed resources are indeed shared.

In `Azureus`, we added reporting of stack-trace information when the assertion is violated, and identified 16 program locations in which disposed resources are shared. Of course, not every shared resource leads to a problem in runtime, and this depends in part on the user interaction with the application (whether the shared resource is indeed used after it has been disposed). We note that this dependency on user interaction makes such bugs very hard to reproduce, and that currently there are several such open bugs in the `Azureues` bug-tracker. Heap assertions make it easy to identify the potential sources of these bugs.

Running the other applications for short user interactions, we also found such suspicious disposal in: `frostwire` (2), `freemind` (2), `tvbrowser` (1), and `rssowl` (3).

Redundant Synchronization for Owned Objects Fearing unexpected effects of concurrency, Java programmers often defensively over-synchronize their programs. A programmer trying to improve concurrency of a given code-base may want to remove redundant synchronization. One common case in which synchronization can be safely removed is when `synchronized` is used on a thread-owned object.

To identify such potential cases, we replace code of the form:

```
synchronized(exp) {
  ...
}
```

with code of the form

```
synchronized(exp) {
 if (Phalanx.dom(Thread.currentThread(),exp))
  Phalanx.warning("synchronization on owned object" + exp);
  ...
}
```

where exp can be any pure expression.

We added such assertions to all points using `synchronize` in our applications. In some applications (notably `jrisk`), automatically adding assertions to all synchronized blocks resulted in assertions added into the main UI event loop. Obviously, this had catastrophic results in terms of performance. Otherwise, when assertions are removed from the main event loop, all applications suffered an observable slowdown, but were still operational.

In several applications we found place that synchronized on thread-owned objects. In `jrisk` we added reporting of stack-trace information and found 12 synchronized blocks where an owned object was used for synchronization. Of course, this does not mean that in different configurations of the system synchronization is not needed, but this can direct

a programmer to potential redundant synchronization in her code.

In `JEdit`, we have identified several places where synchronization is clearly performed on a thread-owned object (by inspecting the code). However, our heap probe checking thread ownership evaluated to false. We used the probe $getThreadReach$ to check what threads are reaching the synchronization object, and found out that the object is reachable from several AWT system threads, which was not at all apparent from the application code.

7. Discussion

Heap queries are an extremely powerful tool for getting global knowledge about program state at runtime. Here we discuss potential improvements for heap queries in terms of efficiency and usability.

7.1 Further Improving Throughput and Pause Times

Section 4 provides the core machinery for efficiently evaluating heap probes, and shows how to benefit from their parallelization. Here, we list some of the ways to further improve the throughput and response times.

Evaluating multiple probes in parallel PHALANX allows multiple probe requests to be issued in parallel. The VM provides many bits for object tagging, so in principle we can evaluate several probes simultaneously using different bits.

Concurrent Evaluation with Snapshots A possible concern for our implementation is the pausing of the application while the probe is evaluated. One solution to this problem is performing a heap snapshot by using a copy-on-write write barrier as in Yuasa's concurrent scheme [36]. In this way, we could evaluate a probe *while* the application continues to execute. One challenge for this approach is dealing with the case where a second probe is started before the first has finished.

Using Local Information The global nature of our probes means that modifying references to objects other than the parameters can affect the result of the probe. One optimization involves remembering local information (e.g., the value of a probe), and updating it locally when the heap changes (e.g., on write barrier). For example, if we are monitoring whether the object is pointed to from the heap, we could set a specific bit in the object header whenever a reference to the object from the heap is created; *cf.* dynamic escape analysis [30]. Although this technique can tell conclusively that an object is not pointed to from the heap, it cannot tell conclusively that it *is* pointed to.

7.2 Expressiveness of Heap Probes

Access to Objects If a probe says that a thread T refers to object O, then, indeed, there is some pointer chain from T to O – *at the level of pointers in the heap*. That does not imply that this chain can actually be followed. For example, if T refers to some object U, which holds a reference to O in a private field that is never exposed, there is no way for T to touch O *in Java*. Even distinguishing between private and non-private fields would not suffice: a standard Java coding style involves private fields `x` and public methods `getX()` for accessing them. Checking statically whether T can get at O in general is undecidable. Checking it efficiently in runtime is a worthy challenge.

Constraining Reachability When debugging or specifying a program with heap probes, it is often necessary to ignore some of the references to get useful information. For example, it may be necessary to ignore `ThreadGroup`s, which provide cross-references between potentially large collections of threads, and can result in irrelevant sharing. Another example is ignoring paths through the `ConnectionManager`'s map from Fig. 1. Our experience to date suggests that it is easy to avoid such references using the parameters `avoid` and `cut` of the probes `getThreadReach` and `reachThrough`, respectively.

Atomicity A single heap probe, by itself, is atomic, e.g., `getThreadReach(t, a)` gives the threads which own `t` at one instant of computation. In some case we need to evaluate several probes atomically, on the same snapshot, for example, to check that the threads referring to `t` are disjoint from those referring to `u`. We can capture the first set of threads with one heap probe, and the second with another – but these probes do not happen on the same snapshot of the heap. The heap may change between the evaluation of two probes; a thread might gain or lose references to `t` or `u` in between, making the disjointness test have the wrong result. We plan to extend our implementation to perform multiple heap probes on a single snapshot of the heap.

Lock Assertions In a concurrent setting, the programmer may design locking mechanisms that require that a specific lock (or set of locks) would be to acquired prior to accessing a field. To support it, the probes can be extended to check that on relevant field accesses the protecting locks are indeed acquired.

7.3 Transfer of Ownership and Uniqueness

In addition to the heap probes of Table 2, we provide an update operation `unique(ref)` that enforces the uniqueness of `ref` as a unique reference to its object by invalidating all other references to it. They are replaced by a special value which, when dereferenced, causes an exception. This is similar to the widely used notion of uniqueness (*e.g.*, [17, 5, 2]), except that we allow the creation of new aliases to the object after the existing ones have been invalidated.

In Fig. 3, we want to transfer ownership of the newly-created socket to the `RequestThread`. The local variable `wsocket` is explicitly set to null so that the `RequestThread` actually does own the socket. Leaving `wsocket` set would violate the invariant, albeit harmlessly because that variable is dead at that point.

We would like to *guarantee* that the ownership of the newly created socket is transferred from the `SimpleWebServer` to the `RequestThread`. In the example of Fig. 3 the programmer has to explicitly set the local variable `wsocket` in `run()` to null such that the `RequestThread` thread would indeed own the socket object when its execution begins.

This could also be accomplished by a call to `unique(socket)` inside of `RequestThread`, instead of the assertion. This approach could be useful to detect, *e.g.*, spurious references to the socket from *inside* the `RequestThread`, as well as from outside.

7.4 Using Heap Probes in Production

Heap probes in our current implementation are excellent tool for testing and debugging, but it is not yet clear whether they are mature enough for deployment in a production system. In principle they could be, since most of the expensive computation in most heap probes can be reused for garbage collection. Several concerns must be addressed before they are production-strength.

Overhead Control PHALANX can use QVM's overhead manager to enforce an overhead limit for evaluating assertions. The QVM overhead manager supports sampling of assertions and can adaptively adjust sampling rates based on the observed time consumed by assertion evaluation. We do not describe this functionality here, as it is not a contribution of this paper, but it can be leveraged in a realistic deployment of heap queries.

Security and privacy Since heap queries are allowed to observe the whole heap, there is a concern that a user can gain access to information that would have been otherwise protected by the Java security system. Note, however, that our current heap probes only allow to reason about the shape relationships between objects and do not allow to examine the actual values stored in an object's fields. The information they reveal may be acceptable for many non-safety-critical systems.

7.5 Integrating Heap Probes with other Tools

When assertions are not sampled, our approach is also applicable for reducing verification efforts. For example, establishing at runtime that parts of the heap are disjoint may guide us to employ more efficient verification techniques that abstract each part of the heap separately.

The heap probes can be extended to provide a comprehensive runtime support for ownership (*e.g.*, the `release` and `capture` operations of [27]). The `unique` operation we provide to enforce reference uniqueness is a first step in that direction.

8. Related Work

QVM This work extends the previous work on QVM [4] in the following aspects. We define a language for specifying heap properties using new heap primitives and set operations. We modified JML compiler for intercepting quantified JML queries and mapping them to PHALANX routines. We support new queries about path properties such as reachability through or avoiding certain objects. We provide parallel algorithms for these queries and also significantly improve the algorithms from [4] by reducing synchronization. We define formal semantics of heap assertions, and show that the parallel algorithms, some of which are quite subtle, correctly implement this semantics. We provide a highly-optimized implementation of all the parallel algorithms on top of QVM. A reference implementation of PHALANX based on JVMTI. This allows us to swap the implementation layer of PHALANX, and use it even without a specialized VM. Of course, we use the specialized VM whenever possible to enjoy the significant performance improvement. We evaluate the scalability of the parallel implementation and compare it to reference implementation with JVMTI. We explore real-world usage scenarios for heap assertions, and demonstrating their usefulness for debugging and program understanding. We evaluate the usefulness of heap assertions in existing applications.

Heap Properties Mitchell [25] provides concise and informative summaries of real world heap graphs arising in production applications. The summaries are done offline and follow a set of useful heuristic patterns for summarizing graphs. In contrast, our goal is to check various user specified heap properties online. [26] studies offline heap snapshots with the goal of finding inefficiencies in memory usage caused by program design.

Chilimbi et. al. [10] provide a two-stage framework suitable for testing, where in the first stage a set of likely heap invariants based on node degree are computed at a small number of program points. Then the instrumented program is executed and checked against these invariants and a bug is reported if a deviation is observed.

SLICK [29] is a runtime tool for checking separation logic specifications of Java programs. It can check that a footprint of the caller's precondition contains the footprint of the callee's precondition, and that methods do not access memory outside of their footprints. It can also check invariants of linked data-structures, *e.g.*, disjointness and reachability through specific fields. In contrast, our tool does not presently distinguish fields, which can be done easily enough from Java. SLICK compiles the specification into Java methods, and instruments the program with calls to these methods. To track the footprints, each object is augmented with an integer "color" field. This approach has higher space overhead than ours; we do not use auxiliary fields. We support thread ownership and other probes that traverse the entire heap, whereas [29] only traverses the portion of the heap reachable within the current method.

Various works have relied on the garbage collector to find memory leaks. Jump et al. [19] use the collector to help in

suggesting potential leaks. Bond et al. [7] studies efficient leak detection for Java. Similarly to us, they make use of available bits in the object header and the adaptive profiling techniques from [16] applied on object use sites, in order to reduce the space and time overheads. This concept could be incorporated into our specialized VM as well. In a recent paper by Aftandilian *et al.* [1], the authors suggest the idea of piggybacking on an existing garbage collector in order to check various heap properties. They propose `isShared` and `isObjectOwned`, but do not provide an implementation of these probes. In contrast, we support heap queries specified in JML, and provide novel parallel algorithms for evaluation of common queries.

Even though it is possible to create an arbitrary aliasing structure, a recent study [15] identifies common patterns of aliasing that arise in practice. These patterns account for nearly all aliasing in their extensive study. We aim to provide programmers with a way to express common patterns and check them during runtime.

The combination of JML and heap primitives allows us to express, and check, a rich range of queries. However, we only provide optimized implementation for some of these queries. In the future, we intend to implement more general query-optimization strategies (e.g., see [12]) into our query translation.

JVMTI JVMTI could also be used to implement heap probes, though it would not work as well as our parallel implementation in the VM. Unfortunately JVMTI does not directly provide heap traversal from a *thread*. Implementing a traversal from a thread using JVMTI hooks is possible but inefficient. This hampers the viability of some of our most important heap probes. Furthermore, JVMTI hides the garbage collector's parallelism from the programmer. Using JVMTI would add extra inefficiency to our probes, which do a great deal of concurrent computation. Working inside the virtual machine allows the use of crucial optimizations that JVMTI resists.

Ownership Control Ownership simplifies reasoning about object-oriented programs by controlling the permitted aliasing. Ownership has been used in many settings. It has been used to ensure representation independence (*e.g.*, [6]), to guarantee thread safety (*e.g.*, [8]), and to enable modular reasoning (*e.g.*, [31]).

A wide variety of static approaches have been proposed for enforcing ownership (*e.g.*, see [11]). These approaches typically impose strict restriction on ownership transfer, requiring that uniqueness holds on transfer (*e.g.*, [17, 5, 2]), or impose a high annotation burden.

We believe that there is a way to express and enforce ownership and sharing properties, without the need to transition into a full-fledged ownership type system. We hope that over time this would lead to better structured code in terms of ownership, and ultimately to the eventual adoption of a proper ownership type system.

Our approach to ownership assertions complements static approaches for enforcing ownership. In particular, our approach may enable a type system to tentatively allow some cases when ownership cannot be established, leaving an ownership check to be performed at runtime. In addition, the VM support for ownership properties can provide an alternative efficient implementation to the runtime support required by some ownership type systems (*e.g.*, [27]).

9. Conclusion

In this paper, we presented PHALANX, a practical tool for dynamically checking expressive heap queries. PHALANX evaluates queries using parallel algorithms, thus leveraging available system cores to speed up evaluation.

PHALANX provides primitives that can be used for reasoning about heap properties in JML annotations. This allows us to harness the full power of JML annotations and use heap queries inside preconditions, postconditions, invariants, and assertions. Common queries are translated by a modified JML compiler into calls to the PHALANX runtime, which evaluates them efficiently using parallel algorithms.

We presented an overall of 7 novel parallel algorithms for heap queries, the algorithms use involved synchronization to efficiently parallelize query evaluation.

We implemented PHALANX on top of a production virtual machine, enabling us to use it on real world applications. We also implemented a portable version of PHALANX using JVMTI. Our evaluation demonstrates that: (i) PHALANX is significantly more efficient than the JVMTI reference implementation; and (ii) parallel query evaluation scales almost linearly with the number of cores.

Our preliminary study shows that heap queries are useful in realistic scenarios for real applications. In particular, we show how heap queries help us to easily detect several cases of suspicious disposal of shared resources, and cases of redundant synchronization over objects that are thread-owned.

We believe that PHALANX greatly increases the value that programmers get from writing expressive assertions, and hope that this additional benefit would lead to wider adoption of annotation-based techniques.

References

[1] AFTANDILIAN, E., AND GUYER, S. Z. Gc assertions: Using the garbage collector to check heap properties. In *MSPC* (2008), ACM.

[2] ALDRICH, J., KOSTADINOV, V., AND CHAMBERS, C. Alias annotations for program understanding. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2002), ACM, pp. 311–330.

[3] ANDERSEN, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Univ. of Copenhagen, May 1994.

[4] ARNOLD, M., VECHEV, M. T., AND YAHAV, E. QVM: an efficient runtime for detecting defects in deployed systems. In *OOPSLA* (2008), pp. 143–162.

[5] BAKER, H. G. 'use-once' variables and linear objects - storage management, reflection and multi-threading. *SIGPLAN Notices 30*, 1 (1995), 45–52.

[6] BANERJEE, A., AND NAUMANN, D. A. Ownership confinement ensures representation independence for object-oriented programs. *J. ACM 52*, 6 (2005), 894–960.

[7] BOND, M. D., AND MCKINLEY, K. S. Bell: bit-encoding online memory leak detection. *SIGOPS Oper. Syst. Rev. 40*, 5 (2006), 61–72.

[8] BOYAPATI, C., LEE, R., AND RINARD, M. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02* (2002), ACM, pp. 211–230.

[9] CALCAGNO, C., DISTEFANO, D., O'HEARN, P., AND YANG, H. Compositional shape analysis by means of bi-abduction. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2009), ACM, pp. 289–300.

[10] CHILIMBI, T. M., AND GANAPATHY, V. Heapmd: identifying heap-based bugs using anomaly detection. vol. 34, ACM, pp. 219–228.

[11] CLARKE, D. G. *Object ownership and containment*. PhD thesis, New South Wales, Australia, Australia, 2003.

[12] DEMSKY, B., CADAR, C., ROY, D., AND RINARD, M. Efficient specification-assisted error localization. In *In Proceedings of the Second International Workshop on Dynamic Analysis* (2004).

[13] DISTEFANO, D., AND J, M. J. P. jstar: towards practical verification for java. In *OOPSLA '08* (New York, NY, USA, 2008), ACM, pp. 213–226.

[14] ECLIPSE. Standard widget toolkit (swt). http://www.eclipse.org/swt/.

[15] HACKETT, B., AND AIKEN, A. How is aliasing used in systems software? In *SIGSOFT FSE* (2006), pp. 69–80.

[16] HAUSWIRTH, M., AND CHILIMBI, T. M. Low-overhead memory leak detection using adaptive statistical profiling. *SIGPLAN Not. 39*, 11 (2004), 156–164.

[17] HOGG, J. Islands: aliasing protection in object-oriented languages. In *OOPSLA '91* (New York, NY, USA, 1991), ACM, pp. 271–285.

[18] Jibble web server. http://www.jibble.org/jibblewebserver.php.

[19] JUMP, M., AND MCKINLEY, K. S. Cork: dynamic memory leak detection for garbage-collected languages. *SIGPLAN Not. 42*, 1 (2007), 31–38.

[20] LEA, D. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Boston, Massachusetts, 2000.

[21] LEAVENS, G. T., CHEON, Y., CLIFTON, C., RUBY, C., AND COK, D. R. How the design of jml accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program. 55*, 1-3 (2005), 185–208.

[22] LEV-AMI, T., AND SAGIV, M. TVLA: A framework for Kleene based static analysis. In *Saskatchewan* (2000), vol. 1824 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 280–301.

[23] LIU, H., AND MOORE, J. S. Java program verification via a jvm deep embedding in acl2. In *TPHOLs* (2004), pp. 184–200.

[24] MICHAEL, M. M., VECHEV, M. T., AND SARASWAT, V. A. Idempotent work stealing. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2008), ACM, pp. 45–54.

[25] MITCHELL, N. The runtime structure of object ownership. In *ECOOP* (2006), D. Thomas, Ed., vol. 4067 of *Lecture Notes in Computer Science*, Springer, pp. 74–98.

[26] MITCHELL, N., AND SEVITSKY, G. The causes of bloat, the limits of health. In *OOPSLA* (2007), R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., Eds., ACM, pp. 245–260.

[27] MÜLLER, P., AND RUDICH, A. Ownership transfer in universe types. In *OOPSLA '07* (New York, NY, USA, 2007), ACM, pp. 461–478.

[28] NAIK, M., AIKEN, A., AND WHALEY, J. Effective static race detection for java. In *PLDI* (2006).

[29] NGUYEN, H. H., KUNCAK, V., AND CHIN, W. N. Runtime checking for separation logic. In *VMCAI* (2008), LNCS.

[30] QIAN, F., AND HENDREN, L. An adaptive, region-based allocator for java. In *Proceedings of the third international symposium on Memory management* (Jun 2002), ACM Press, pp. 127–138.

[31] RINETZKY, N., POETZSCH-HEFFTER, A., RAMALINGAM, G., SAGIV, M., AND YAHAV, E. Modular shape analysis for dynamically encapsulated programs. In *ESOP* (2007), pp. 220–236.

[32] RODITTY, L. A faster and simpler fully dynamic transitive closure. In *SODA '03* (Philadelphia, PA, USA, 2003), Society for Industrial and Applied Mathematics, pp. 404–412.

[33] SAGIV, M., REPS, T., AND WILHELM, R. Parametric shape analysis via 3-valued logic. *ACM Trans. on Prog. Lang. and Systems (TOPLAS) 24*, 3 (2002), 217–298.

[34] STEENSGAARD, B. Points-to analysis in almost-linear time. pp. 32–41.

[35] YANG, H., LEE, O., BERDINE, J., CALCAGNO, C., COOK, B., DISTEFANO, D., AND O'HEARN, P. W. Scalable shape analysis for systems code. In *CAV* (2008), pp. 385–398.

[36] YUASA, T. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software 11*, 3 (Mar. 1990), 181–198.

[37] ZEE, K., KUNCAK, V., AND RINARD, M. Full functional verification of linked data structures. In *ACM Conf. Programming Language Design and Implementation (PLDI)* (2008).