PHALANX: Parallel Checking of Expressive Heap Assertions

Martin Vechev
IBM Research

Eran Yahav
IBM Research

Greta Yorsh
IBM Research

Unrestricted use of heap pointers makes software systems difficult to understand and to debug. To address this challenge, we developed PHALANX — a practical framework for dynamically checking expressive heap properties such as ownership, sharing and reachability. PHALANX uses novel parallel algorithms to efficiently check a wide range of heap properties utilizing the available cores.

PHALANX runtime is implemented on top of IBM's Java production virtual machine. This has enabled us to apply our new techniques to real world software. We checked expressive heap properties in various scenarios and found the runtime support to be valuable for debugging and program understanding. Further, our experimental results on DaCapo and other benchmarks 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.

To encourage adoption by programmers, we extended an existing JML compiler to translate expressive JML assertions about the heap into their efficient implementation provided by PHALANX. To debug her program, a programmer can annotate it with expressive heap assertions in JML, that are efficiently checked by PHALANX.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]; D.2.4 [*Software/Program Verification*]: Assertion checkers

General Terms Algorithms, Reliability

Keywords ownership, parallel garbage collector, virtual machine

1. Introduction

Modern programming languages increase programmer productivity by providing correctness-checking mechanisms for common low-level programming errors (e.g., dereferencing of a null pointer). However, such low-level errors are often only a symptom of mishandling of pointer aliasing. Incidental and accidental pointer aliasing result in unexpected side effects of seemingly unrelated operations, and are a major source of system failures. This problem is exacerbated in the presence of concurrency.

Despite significant advances in checking and verification of heap properties, practical adoption of such approaches remains limited. Static approaches are either imprecise (e.g., [3, 28]), do not scale to large applications (e.g., [13, 26, 29]), or focus on specific properties (e.g., [9]). Most of these approaches do not handle concurrent programs. Dynamic approaches are more promising in terms of scaling, but are either limited to local properties [10], require heap snapshots to be analyzed offline [21], or support only a limited set of simple heap queries [1, 4]. Specification languages for Java such as JML [18] allow the programmer to specify expressive heap assertions, but lack runtime support for checking them.

Ownership types hold great promise, as they simplify reasoning about object-oriented programs by controlling the permitted aliasing. A wide variety of static approaches have been proposed for enforcing ownership (e.g., [11]). These approaches typically impose strict restriction on ownership transfer, requiring that uniqueness holds on transfer (e.g., [2, 5, 16]), or impose a high annotation burden. These factors have been an obstacle for wide adoption of ownership types in practice.

We present PHALANX — a practical tool for dynamically checking expressive heap properties that involve ownership, sharing and reachability. Our approach complements existing static approaches for enforcing ownership, such as ownership types (e.g., [8, 11, 12, 14, 23]), and may enable their wider adoption. 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, our approach can provide an efficient alternative runtime support of ownership, required by some ownership type systems (*e.g.*, [23]).

With the advent of multicore systems, we envision that some cores could be dedicated to performing software quality tasks as part of extended language runtime. In particular, it will enable checking properties that have been traditionally considered too expensive even for debugging scenarios. This work makes a step in that general direction by extending language runtime support to a new level of expressivity.

Expressive Query Language 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. Efficient runtime support of common heap queries enables the programmer to check properties that are difficult or practically impossible to check otherwise, such as properties like sharing 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 Sec. 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 ownership and other queries useful in practice. This dictates the need for efficient support of these refined queries.

To make the deployment of 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 [18]. We provide a

ISMM'10, June 5–6, 2010, Toronto, Ontario, Canada.

modified version of the JML compiler to translate heap queries to their efficient implementation in the PHALANX runtime.

To invoke a heap query, the program holds pointers to the relevant objects in the heap. The user-intended meaning of the query usually does not 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 to prove that the algorithms implemented in the runtime compute the intended results.

Efficient Runtime Support We provide efficient algorithms for evaluating various heap queries, using a traversal over the program heap. Much of the machinery necessary for evaluating these heap queries is already available in modern virtual machines (VM), and the tracing garbage collector (GC) makes use of this machinery when traversing the heap. However, many useful heap queries cannot be directly checked using the information obtained during GC. Intuitively, a tracing GC computes reachability (transitive closure) information, which tells us *whether* an object is reachable or not, but it does not tell us *how* an object can be reached. The latter information is required for checking path properties such as domination, ownership and disjointness.

Every one of the common queries are evaluated using a single linear-time traversal of the heap, matching the complexity of GC traversal, and keeping auxiliary space to a minimum. This is particularly challenging for queries such as disjointness and ownership. Moreover, we provide parallel versions of these algorithms designed to take advantage of the available cores to speed up query evaluation.

The overhead of query evaluation is highly dependent on how many assertions are checked, where the assertions are in the program, the shape (structure) of the heap, and the result of the query (e.g., query checking whether an object is shared must traverse the entire heap to return a negative answer, but when it returns a positive answer, it often completes very fast).

Production Virtual Machine Our algorithms are implemented in IBM's Java production virtual machine enabling us to achieve efficiency that is very hard to obtain by other means. Further, the integration with the production VM enables us to easily 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 number of potential sources of bugs and inefficiencies.

The main 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 heap queries to be useful.
- A small set of natural primitives that can be used to express many heap queries as 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 as well as real-world applications, including a comparison of the parallel implementation in a production VM to a reference implementation outside of the VM that is based on JVMTI.

This work enables checking of heap assertions beyond those expressed and checked by existing systems. We address all aspects of assertion checking, from user interaction in JML to efficient runtime implementation, assisting the user in the difficult task of debugging large-scale applications.

1.1 Related Work

In this section we briefly discuss the most closely-related work.

QVM This work extends QVM [4] in the following aspects. 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. We explore real-world usage scenarios for heap assertions, and demonstrate their usefulness for debugging and program understanding. We evaluate the usefulness of heap assertions in existing applications. We evaluate the scalability of the parallel implementation and compare it to reference implementation with JVMTI. We define a language for specifying heap properties using new heap primitives and set operations. We modified JML compiler to intercept quantified JML queries and map them to PHALANX methods.

Heap Properties Mitchell [21] provides concise and informative summaries of real world heap graphs arising in production applications. The summaries are computed 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. [22] studies offline heap snapshots with the goal of finding inefficiencies in memory usage caused by program design.

Various works have relied on the GC to find memory leaks. Jump and McKinley [17] use the collector to help in suggesting potential leaks. Shacham et. al [27] use the collector to track data-structure utilization and identify bloat in Java collections. Bond and McKinley [7] study efficient leak detection for Java.They make use of runtime support and adaptive profiling techniques from [15] applied on object use sites, in order to reduce the space and time overheads. This concept can be used in PHALANX as well.

In [1], the authors suggest the idea of "piggybacking on a GC traversal" to check various heap properties. That is, the method of [1] reuses the work done by GC traversal to evaluate queries about the heap. In our work, we do the opposite: GC traversal can reuse the work performed by query evaluation. Note that we do reuse GC traversal *components* to implement query evaluation.

In [1], the main concern is what properties can be checked by piggybacking on a GC traversal. Our concern is different, namely, to check expressive path-based queries such as ownership and disjointness. Reusing the work of GC for query evaluation, as done in previous work [1], is inherently limiting the kinds of queries that can be computed: as we explained earlier, GC traversal only computes *whether* an object is reachable, while checking properties such as ownership requires information on *how* the object is reachable, via what paths. Our experience has revealed the need for more expressive queries, such as ownership, disjointness, and reachability through or avoiding certain objects. Such queries cannot be piggybacked on GC traversal, but our algorithms are designed such that GC traversal can reuse the traversal work done by query evaluation. Further compared to [1], we support heap queries specified in JML and provide novel parallel algorithms with fine-grained synchronization for evaluation of common queries.

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

Figure 1. Code fragment from JdbF.

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 Sec. 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 [24], 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` object. 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*. 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.

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.

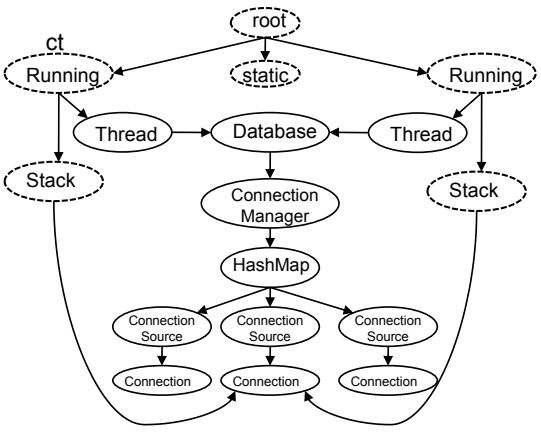

Figure 2. Example graph that represents a state of the program JdbF from Fig. 1. We use dashed line for the special nodes, which do not represent heap-allocated objects. All other nodes represent objects labeled with their (short) class name. We omit some details of `Map` implementation for clarity.

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, which has access to the VM's internal information. Finally, PHALANX evaluates this assertion atomically: all subexpressions are evaluated on the same snapshot of the heap and the application threads are suspended during 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 reference fields while the assertion is being evaluated, 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 shown in Fig. 2. Such path properties go beyond 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, the assertion can be expressed in JML (see Fig. 1, lines 5-6).

In Sec. 4, we describe a number of efficient algorithms for evaluating such heap queries. All of our algorithms have corresponding efficient parallel versions, enabling optimal usage of the underlying multi-core machine. The algorithms are not specific to our setup, and can be implemented in any VM.

3. Expressive Language of Heap Queries

JML assertion language [18] 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. We also benefit from JML's support for method specification, specification inheritance, invariant checking, etc.

In this section, we start by defining the meaning of heap queries. We present the core 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

To define the meaning of heap queries, we represent Java program states as graphs. It allows us to define the semantics using standard set and graph operations. It also simplifies the reasoning about correctness of our parallel implementation of the common heap queries, described in Sec. 4.

Graph definition $g(s)$***:*** For a state s, 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 $g(s)$. Graph $g(s)$ is 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 $L\colon V \to \mathit{Classes} \cup \{\mathit{root}, \mathit{stack}, \mathit{running}, \mathit{static}\}$ is the node-labeling function.

Graph Notations We now introduce some standard notations for the graph operations which we use in the definition of the primitives. Let $g(s) = \langle V, E, L\rangle$ be a graph as above. 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 $\mathit{src}(\pi)$ and $\mathit{dst}(\pi)$ to denote the first and last nodes of the path π. For a path π, we use $\mathit{nodes}(\pi)$ 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', defined by $\Pi_{g(s)}(v, v') \stackrel{\text{def}}{=} \{\pi \in \Pi \mid \mathit{src}(\pi) = v, \mathit{dst}(\pi) = v'\}$.

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 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`*. 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. 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* `HashMap` *object. Furthermore, both running threads refer to the same* `Connection` *object from their stack.*

Heap Primitives Provided by PHALANX The primitives supported by PHALANX runtime and their semantics are shown in Table 1. The primitive `dom` returns `boolean`. All the other 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 represents the object pointed to by `o` in state s. Similarly, the semantics of a parameter `a` 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. *Suppose that the following JML assertion is added to the program of Fig. 1, after line* `4:` `pred(c).size() == 1`*. It uses the primitive* `pred` *to check the* `Connection` *object pointed to by* `c` *has exactly one predecessor. This assertion is satisfied in the state shown in Fig. 2 because only a single object in the heap, of type* `ConnectionSource`*, has a field pointing to* `c`*.*

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 would hold.*

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 (unlike ownership defined in [12]). The semantics does consider paths that go through stacks of other running threads as violating domination. This allows us to check ownership properties pertaining to synchronization between threads.

3.2 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 of Table 2 lists names of corresponding heap probes — PHALANX runtime methods that efficiently implement these queries, as described in Sec. 4. For example, object-ownership query `dom(x,y)` is implemented by `isObjectOwned` probe. Thread-ownership is just a special case of object-ownership where the owner is `Thread.currentThread()`, the current thread object. Queries related to threads (e.g., getThreadReach and isThread-Owned) 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.

Along with the heap primitives shown in Table 1, PHALANX provides a useful shorthand that returns all the "roots" of a given thread; `roots(Thread t)` returns the JMLObjectSet which consists of all objects pointed from the thread's stack and the thread object itself: `roots(Thread t)` $\stackrel{\text{def}}{=}$ `stack(t).insert(t)`.

4. Heap Probe Algorithms

In this section, we present the algorithms for computing the heap probes listed in Table 2. We begin by discussing the design decisions we made to create efficient algorithms.

Low-overhead Evaluation of heap queries involves computing transitive closure of the heap graph. Transitive closure can be precomputed in $O(V \times (V+E))$ operations (for a heap with V objects and E references), and updated using incremental algorithms (*e.g.*, Roditty [25]). 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, using a heap traversal. The worst-case time complexity for all our algorithms is $O(V + E)$ operations for a heap with V objects and E references. Evaluating P of them could take $O(P \times (V + E))$ operations. However, in practice, we expect $P \ll V$ and heap queries are much less frequent than heap updates, and hence for large program heaps, this approach is likely to be superior to

Name	Semantics	Description
`running()`	$\{r.obj \mid L(r) = \textit{running}\}$	Running threads
`stack(Thread t)`	$\{v \mid (r.stack, v) \in E \wedge L(r) = \textit{running} \wedge r.obj = \mathtt{t}\}$	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)`	$\{v \mid \Pi(\mathtt{o}, v) \neq \emptyset\}$	Objects reachable from object `o`
`reach(Object o,Object[] a)`	$\{v \mid \exists \pi \in \Pi(\mathtt{o}, v).\textit{nodes}(\pi) \cap \mathtt{a} = \emptyset\}$	Objects reachable from object `o` without going through any of the objects in `a`
`pred(Object o)`	$\{v \mid (v, \mathtt{o}) \in E \wedge L(v) \in \textit{Classes} \wedge \Pi(\textit{root}, v) \neq \emptyset\}$	Reachable objects pointing to object `o`
`dom(Object o1,Object o2)`	$\exists \pi \in \Pi(\mathtt{o1}, \mathtt{o2}) \wedge$ $\forall \pi \in \Pi(\textit{root}, \mathtt{o2}).\mathtt{o1} \in \textit{nodes}(\pi) \vee \textit{ct.stack} \in \textit{nodes}(\pi)$	There is a path from `o1` to `o2`, and every path from root to `o2` goes 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 no 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(o1, o2)`	Does the object o_1 dominate o_2?	isObjectOwned(Object o_1, Object o_2)
`dom(Thread.currentThread(), o)`	Does the current thread's object dominate o?	isObjectOwned(Object o_1, Object o_2) (where o_1 is `currentThread` and o_2 is o)
`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.

incrementally updating a pre-computed transitive closure when the heap is modified. In practice, constants matter and it is important to perform as few passes over the heap as possible.

Portability Our algorithms are not specific to a particular VM and hence can be implemented in any VM desiring support for heap assertions, e.g. Javascript, C#, etc. The algorithms are designed to use standard components already present in these VMs, such as heap traversal machinery, synchronization primitives, auxiliary object data. The algorithms can take advantage of existing efficient implementation of these components, e.g., load balancing [20], and future advances in their technology.

Moreover, we designed the algorithms in a way where any work done for query evaluation can be reused for garbage collection. Note that the converse is *not* true: many probes cannot be implemented by only relying on information computed by the GC. In this section, we present the algorithms independently of our runtime. In Sec. 5, we discuss various challenges in integrating them into a production virtual machine.

We designed the algorithms to leverage available cores in the system for speeding up heap queries. Our algorithms operate efficiently in both sequential and parallel settings. A single heap query can be computed by a single thread or by multiple threads, if available. Our algorithms operate by stopping the application, evaluating the heap query on the program heap, and resuming the application. Fig. 4 shows the pseudo-code, including synchronization.

We first review the standard heap traversal components (Sec. 4.1), then we explain how our query evaluation algorithms work in a sequential setting (Sec. 4.2), and finally we discuss the necessary synchronization for ensuring correct and efficient parallel query evaluation (Sec. 4.3).

4.1 Heap Traversal Components

Fig. 3 shows the standard heap traversal components provided by most virtual machines.

The set T_a denotes the set of (running) application threads in the system at the time a probe is invoked. For an application thread t_a, we use $t_a.stack$ and $t_a.obj$ to denote the thread's stack and object.

The procedure `mark-thread()` marks the objects directly pointed to by an application (running) thread's stack and thread's object, but does not perform further traversal from that set. The procedure `trace()` performs heap traversal to compute the set of objects reachable from the set $pending$. 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 $true$. The default implementation of `trace-step` is empty.

4.2 Sequential Algorithms

Fig. 4 shows the parallel algorithms for our heap queries. The algorithms are designed in a way where one can simply elide all synchronization constructs (explained in Sec. 4.3) and easily obtain a correct and efficient sequential algorithm. This is useful in settings where only a single core is dedicated to heap query evaluation. In this section, we explain how the algorithms work when only a single thread evaluates the query. In the next section, we discuss the challenges with parallel evaluation.

isThreadOwned The implementation of `isThreadOwned(`t_a`, o)` is shown in Fig. 4(b). 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*. 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 similar to tracing collectors, with the key difference being the specific order in which threads are processed. If we would like to perform garbage collection after this probe completes, we can proceed to mark and trace from the roots of the current application thread t_a. That is, collection can reuse the work that was done for the probe.

isObjectOwned The implementation of `isObjectOwned(src, tgt)` is shown in Fig. 4(d). Recall that this probe only returns *true* when all paths to tgt go through src 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 src without tracing from it, and then trace through all other roots. Since src is marked during tracing, it will not be traced through and all objects that are reachable only through src will remain unmarked.

First, the algorithm uses `tag-object()` to mark the src object *without tracing from it*. Then, the algorithm switches to $skip$ phase and marks all objects pointed to from the roots (except tgt) without tracing from them yet. The purpose of this phase is to avoid marking tgt if it is pointed directly from the roots as we want to reason only about heap paths. Then, the phase is switched to $none$ to perform tracing as usual. During tracing, if the object src is encountered, tracing will not continue to trace from it because it is already in the $Marked$ set. Upon completion of the tracing phase, we check whether tgt is marked. If it is marked, the probe returns *false*. Otherwise, if tgt is reachable from src, it returns *true*.

We need to manage the object src carefully, to allow garbage collection to be performed during this probe. The src object is marked during the probe but it is not placed in the $pending$ set. If we continue with a garbage collection after this probe, we need to make sure that src is added to $pending$.

One of the challenges of implementing probes of this type in a VM is dealing with stack pointers. In particular, objects src and tgt 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 from the current thread to tgt. Alternative implementations could identify which stack pointers to consider and which stack pointers to ignore, but this is very challenging in practice, especially due to JIT optimizations.

getThreadReach The implementation of `getThreadReach(o, avoid)` is shown in Fig. 4(f). 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 the $avoid$ set. Then we compute the transitive closure from that thread. If after that, o is marked, then the application thread is inserted into the $result$ set, otherwise, we do not insert it. After processing each application thread, the $Marked$ set is initialized to $\emptyset$.

This probe tracks reachability from both thread stacks and thread objects. In Table 2, we specialize it further to track reachability only from thread stacks or only from thread objects.

reachThrough The implementation of `reachThrough(o1, cut, o2)` is shown in Fig. 4(e). 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 objects in 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. 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 any object in the set cut. In that case, the probe returns *false*. Otherwise, the probe returns *true*.

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

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

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

push-object(o)
  pending ← pending ∪ {o}

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

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

Figure 3. Basic Components for Heap Traversal

4.3 Synchronization

To evaluate heap queries in parallel, we use a set of *evaluator threads*. Theoretically, the algorithm places no limits on the number of these threads, although in practice, we can create one thread for each available processor. To ensure correct operation when multiple threads are evaluating the heap query, the algorithms make careful use of the following synchronization primitives:

Barrier: A barrier is provided by `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 evaluator threads are released to continue.

Barrier with Master thread: 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.

In the rest of this section, we explain the synchronization used in the algorithms in Fig. 4. The variables $result$, $phase$, $Owned$, $Marked$, T_a are shared, all other variables are local to the evaluator thread. To avoid clutter, we abstract away the internal load balancing mechanism of the VM with the $pending$ set. That is, the VM may transparently move objects from one local $pending$ set to another, without affecting the correctness of the algorithms.

isReachable In the algorithm of Fig. 4(a), every evaluation thread marks the `src` object, and then traces from it. All evaluation threads eventually block at `barrier-and-release-master()`. When this happens, the object pointed to by `tgt` is guaranteed to be marked if and only if it is reachable from `src`. At this point, the master thread sets the return value based on whether the `tgt` object is marked. This need only be done by one thread, hence the use of the master. Then the master releases the other evaluator threads. The algorithms for `isThreadOwned`, `isShared`, and `getThreadReach` use barriers in a similar way.

isDisjoint In Fig. 4(g), in the beginning, every evaluator thread sets the shared $result$ variable to *true* and the shared $phase$ variable to $dual$. After tracing, all evaluator threads synchronize via the barrier to ensure completion of the $dual$ phase, before switching

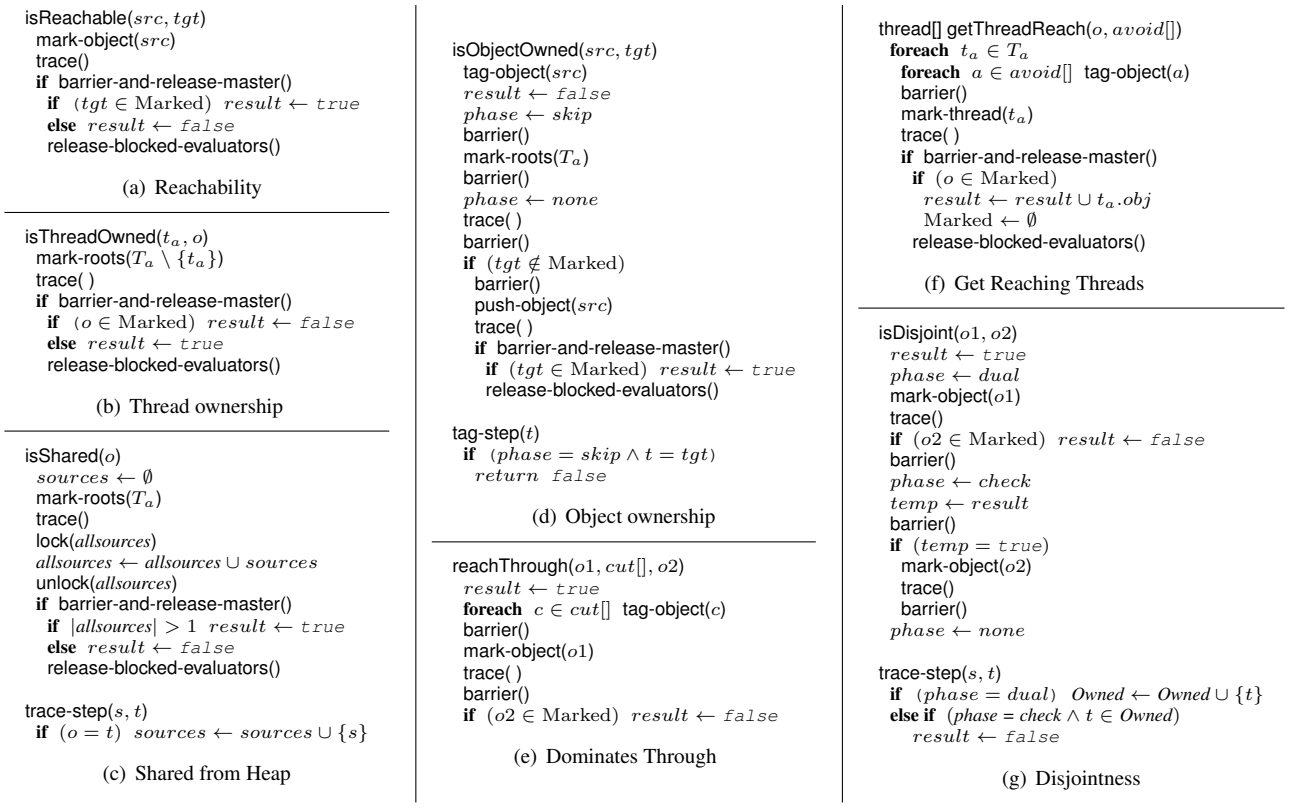

Figure 4. New parallel algorithms for evaluating common heap queries.

to the $check$ phase. After synchronously switching to the $check$ phase, every evaluator thread reads the value of $result$ into its local variable $temp$. If after the barrier the result is still *true*, the evaluator starts tracing from $o2$. Upon completion, the evaluator threads synchronize and switch the phase to $none$. Barriers are similarly used for switching phases in `isObjectOwned`.

isShared In Fig. 4(c), every evaluator thread uses a private set $sources$ to record the objects pointing to o that it encountered during its traversal. By using private sets, we avoid 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 lock. Combining the local sets is required as it is possible that o is shared but each parallel evaluator reached it only once during its traversal. Finally, the threads synchronize, the master thread computes the result (if there is more than one object in $allsource$, we return *true*, otherwise *false*) and releases the rest of the evaluator threads.

For clarity of presentation we do not include some optimizations. For example, if we do not require garbage collection to start after probe computation, in `isDisjoint`, when $result$ is set to *false* by an evaluator thread, the probe can immediately return *true*.

5. Integration

We have extended IBM's Java production virtual machine with our heap probe algorithms. Production grade VMs are complex pieces of code, and correctly implementing the desired heap query semantics is quite challenging. In this section, we briefly describe some of the practical integration issues, which we believe are useful to anyone implementing these algorithms in other VMs.

5.1 Reuse of Infrastructure

Our VM already provides the basic components (Fig. 3) and synchronization primitives (Sec. 4.3).

Evaluator Threads In our VM, when the application starts, evaluator threads are created, one for each core. These evaluator threads are initially blocked. The runtime uses these threads for parallel garbage collection. However, we can reuse these threads for evaluation of our heap queries.

Object Sets Our VM 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 algorithms, when necessary, we use this capability to create and manipulate other sets. For example, in `isDisjoint`, we use the set $Owned$ in addition to $Marked$ and in `getThreadReach`, we can reuse the machinery that sets $Marked$ to be the empty set. In our setting, this is possible to do efficiently as marked bits for each object can reside in a contiguous memory region outside of the actual object space itself, making it easy to re-initialize that marked set.

5.2 Garbage Collection Interaction

Sharing Heap Traversal Components Components used by our heap probes are also used by the garbage collection algorithms. Care must be taken to ensure that changes to these components do not affect the operation of the normal collector. For example, as mentioned in Sec. 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 would be to add arguments to existing internal functions, but this would have spanned changes to a large number of modules, making the implementation less self-contained and more error-prone.

GC during Heap Query Evaluation Our parallel algorithms are specifically designed in a way that the work done during query evaluation can be re-used by the GC. Hence, in addition to answering the required heap query, we have the option to perform garbage collection right after the heap probe finishes, *leveraging the work done by the probe*. That is, the collector can piggyback on our heap query evaluation. Note that the opposite, computing results for our probes *only* based on existing garbage collection is not possible as our probes often require more involved computations over the heap.

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

If we do not wish to perform GC after the heap probe, then the probe returns immediately. In this case, we make sure that all intermediate state used by the probe is reset.

5.3 Integration with the JML Compiler

The original JML compiler does not produce 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 modified the JML compiler to identify common queries and translate them into calls to heap probes supported by PHALANX.

We managed to keep our changes to a minimum by modifying the translation step in the compiler, where Java code is generated from JML expressions. In this phase, we use simple pattern matching on the AST to identify quantified expressions that match one of the common heap queries, and replace each quantified expression by a method call expression invoking the appropriate probe method in 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.

6. Evaluation

In this section, we discuss the effectiveness of our tool when applied to a variety of real world applications. Then, we evaluate the scalability of the parallel algorithms on the popular DaCapo benchmark suite [6].

6.1 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 only to measure the total time that the assertions take is also not ideal, as 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 example, when an object is not shared, evaluating `isShared` on this object 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., concurrency, I/O).

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

To eliminate these factors, we conducted a controlled evaluation of performance with the synthetic benchmarks described in Sec. 6.2. 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.

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

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.

TVLA TVLA [19] is a parametric framework for shape analysis that can be instantiated to create different kinds of analyzers. To reduce 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, maintains a pending list of structures to process with only the structures that changed. We added assertions to check that the pending list is correctly manipulated.

Automatically Added Assertions

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 real-world applications, and in these applications we added direct calls to PHALANX probes.

Most of the applications we considered are interactive ones, thus the specific probes executed 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). 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)) warning(exp);
exp.dispose();
```

We ran our benchmarks with the added assertions. For the most part, the frequency of resource disposal is low enough that the application exhibits very little observable slowdown when running on our heap query enabled VM. 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 at 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 performance 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){...}` where `exp` can be any pure expression, with code of the form

```
synchronized(exp) {
 if (Phalanx.dom(Thread.currentThread(),exp)) warning(exp);
 ... }
```

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 places 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 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.

num. cores	1	2	4	8
antlr	1.9	1.9	1.9	1.9
bloat	7.2	4.2	2.9	2.7
chart	1.1	1.0	1.0	1.0
eclipse	1.0	1.0	1.0	1.0
fop	1.1	1.0	1.0	1.0
hsqldb	14.7	9.8	6.3	4.3
jython	1.6	1.4	1.2	1.2
luindex	3.3	2.5	2.1	1.9
lusearch	9.9	10.0	10.2	9.4
pmd	1.2	1.1	1.1	1.1

Table 4. Slowdown for DaCapo with varying number of cores.

6.2 Performance Evaluation

All of our experiments were conducted on an 8-core 2.4 GHz AMD Opteron running Red Hat Linux with a 1.6.0 IBM Java VM.

For all of our heap probes, we performed a variety of experiments with synthetic benchmarks on large and complex heaps with growing number of cores dedicated to the evaluation of the heap probe algorithm. Typically, when it had to traverse the entire heap (e.g., when `isShared` returned *false*) the scalability of the parallel algorithm was similar to that of the underlying parallel garbage collector. This is to be expected as the probes are enabled to reuse the same techniques for scalability (e.g. work stealing) as the underlying collector.

JVMTI In addition to our parallel heap probes, we also implemented a reference library using the JVMTI heap traversal interface. JVMTI is a powerful native programming interface for use by tools that need to access the JVM state for profiling, debugging, and monitoring. Unfortunately, the JVMTI interface does not allow for *parallel* heap traversal. We compare the performance of our algorithms running with one evaluation thread (e.g. one core) vs. JVMTI. JVMTI implementation was often suffering over 2x slowdowns. This is to be expected as every time an edge is traversed, JVMTI makes an external callback, which is avoided when the algorithm is implemented within the VM. Based on this experience, we did not use the JVMTI library further in any of our experiments.

DaCapo To evaluate the scalability of our assertions on a set of known benchmarks, we chose the popular DaCapo benchmark suite [6]. We instrumented each benchmark in the suite to check the `isShared` heap property and measured the time it took to execute the benchmark with the instrumented code and the time it took to execute the benchmark with the base version without any instrumentation. We ran each benchmark with the large input size option. For a given number of cores, Table 4 reports the slowdowns for the instrumented time over the base time.

As noted earlier, adding meaningful heap-assertions to existing realistic applications such as the DaCapo benchmarks is very challenging. When instrumenting these benchmarks we followed a general instrumentation scheme. For the benchmarks `antlr`, `jython`, and `pmd`, that operate on ASTs, we checked that AST nodes are not shared in some key operations performed on the AST. For `bloat`,

we check sharing of arguments when constructing the SSA representation. For benchmarks such as `chart`, which dispose SWT resources, we checked that the resource is not shared before disposing it. For `eclipse` we added assertions in the DaCapo plugin. For `lusearch` and `luindex`, we checked whether parameters to the search/index methods are shared. For `hsqldb` we check sharing of the parameters to client threads.

The results for `bloat`, `hsqldb`, `luindex`, and `jython` show the benefits of parallelization. As the number of cores increases, the running time improves, getting closer to the base time. The results indicate that for all benchmarks but `hsqldb`, four cores are sufficient to extract the parallelization benefits and adding more cores has little to no effect.

For `antlr` and `lusearch`, adding more cores had little effect on the probe evaluation. A possible reason is that heap graphs encountered at probe evaluation are inherently non-scalable (e.g. a single linked list). For most of the benchmarks, e.g. `fop`, `eclipse`, `chart`, and `pmd`, the overhead of the heap probe is relatively low or non-observable as the number of executed probes with our instrumentation was very low.

7. Conclusion

In this paper, we presented PHALANX, a practical tool for dynamically checking expressive heap queries. PHALANX is implemented on top of a production Java virtual machine, enabling the use of heap queries on real world applications.

To facilitate deployment by programmers, we extended the JML compiler to make use of these queries: developers can use JML annotations to effectively reason about heap properties. 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 PHALANX runtime, which uses efficient parallel algorithms to evaluate them.

Our preliminary study shows that heap queries are useful for real applications. 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.

By addressing all aspects of checking of expressive heap assertions, from user interface in JML to efficient runtime implementation, we have provided a complete setup, making it simpler for programmers to write expressive assertions. We believe that this will enable the wider adoption of practical assertion-based techniques.

Acknowledgements We thank Bard Bloom for his contributions to early versions of this paper. We thank Emery Berger and Peter Müller for their feedback on this work. We also thank the anonymous reviewers for many helpful comments that improved the presentation.

References

[1] AFTANDILIAN, E., AND GUYER, S. Z. GC assertions: using the garbage collector to check heap properties. In *PLDI* (2009).

[2] ALDRICH, J., KOSTADINOV, V., AND CHAMBERS, C. Alias annotations for program understanding. In *OOPSLA* (2002).

[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).

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

[6] BLACKBURN, S. M., GARNER, R., HOFFMAN, C., KHAN, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA* (2006), pp. 169–190.

[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., LISKOV, B., AND SHRIRA, L. Ownership types for object encapsulation. In *POPL* (2003), pp. 213–223.

[9] CALCAGNO, C., DISTEFANO, D., O'HEARN, P., AND YANG, H. Compositional shape analysis by means of bi-abduction. In *POPL* (2009), pp. 289–300.

[10] CHILIMBI, T. M., AND GANAPATHY, V. HeapMD: identifying heap-based bugs using anomaly detection. In *ASPLOS* (2006), pp. 219–228.

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

[12] CLARKE, D. G., POTTER, J. M., AND NOBLE, J. Ownership types for flexible alias protection. In *OOPSLA* (1998), pp. 48–64.

[13] DISTEFANO, D., AND PARKINSON, M. J. jStar: towards practical verification for Java. In *OOPSLA '08* (2008), pp. 213–226.

[14] GROTHOFF, C., PALSBERG, J., AND VITEK, J. Encapsulating objects with confined types. *ACM Trans. Prog. Lang. Syst. 29*, 6 (2007), 32.

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

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

[17] JUMP, M., AND MCKINLEY, K. S. Cork: dynamic memory leak detection for garbage-collected languages. In *POPL* (2007), pp. 31–38.

[18] 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.

[19] LEV-AMI, T., AND SAGIV, M. TVLA: A framework for Kleene based static analysis. In *SAS* (2000), vol. 1824, pp. 280–301.

[20] MICHAEL, M. M., VECHEV, M. T., AND SARASWAT, V. A. Idempotent work stealing. In *PPoPP '09* (2008), pp. 45–54.

[21] MITCHELL, N. The runtime structure of object ownership. In *ECOOP* (2006), pp. 74–98.

[22] MITCHELL, N., AND SEVITSKY, G. The causes of bloat, the limits of health. In *OOPSLA* (2007), pp. 245–260.

[23] MÜLLER, P., AND RUDICH, A. Ownership transfer in universe types. In *OOPSLA* (2007), pp. 461–478.

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

[25] RODITTY, L. A faster and simpler fully dynamic transitive closure. In *SODA* (2003), pp. 404–412.

[26] SAGIV, M., REPS, T., AND WILHELM, R. Parametric shape analysis via 3-valued logic. *(TOPLAS) 24*, 3 (2002), 217–298.

[27] SHACHAM, O., VECHEV, M., AND YAHAV, E. Chameleon: adaptive selection of collections. In *PLDI* (2009), pp. 408–418.

[28] STEENSGAARD, B. Points-to analysis in almost linear time. In *POPL* (1996), pp. 32–41.

[29] 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.