Estimating the impact of heap liveness information on space consumption in Java

Conference Paper *in* ACM SIGPLAN Notices · June 2002

DOI: 10.1145/773039.512437 · Source: DBLP

3 authors, including:

Ran Shaham

15 PUBLICATIONS **391** CITATIONS

SEE PROFILE

Hillel Kolodner

IBM

79 PUBLICATIONS **1,144** CITATIONS

SEE PROFILE

Estimating the Impact of Heap Liveness Information on Space Consumption in Java

Ran Shaham
Tel-Aviv University and
IBM Haifa Research
Laboratory
rans@math.tau.ac.il

Elliot K. Kolodner
IBM Haifa Research
Laboratory
kolodner@il.ibm.com

Mooly Sagiv
Tel-Aviv University
sagiv@math.tau.ac.il

ABSTRACT

We study the potential impact of different kinds of liveness information on the space consumption of a program in a garbage collected environment, specifically for Java. The idea is to measure the time difference between the actual time an object is collected by the garbage collector (GC) and the potential earliest time an object could be collected assuming liveness information were available. We focus on the following kinds of liveness information: (i) stack reference liveness (local reference variable liveness in Java), (ii) global reference liveness (static reference variable liveness in Java), (iii) heap reference liveness (instance reference variable liveness or array reference liveness in Java), and (vi) any combination of (i)-(iii). We also provide some insights on the kind of interface between a compiler and GC that could achieve these potential savings.

The Java Virtual Machine (JVM) was instrumented to measure (dynamic) liveness information. Experimental results are given for 10 benchmarks, including 5 of the SPECjvm98 benchmark suite. We show that in general stack reference liveness may yield small benefits, global reference liveness combined with stack reference liveness may yield medium benefits, and heap reference liveness yields the largest potential benefit. Specifically, for heap reference liveness we measure an average potential savings of 39% using an interface with complete liveness information, and an average savings of 15% using a more restricted interface.

Categories and Subject Descriptors

C.4 [**Performance Of Systems**]: Measurement techniques; C.4 [**Performance Of Systems**]: Performance attributes; D.3.4 [**Programming Languages**]: Processors—*Compilers, Memory management (garbage collection), Optimization, Run-time environments*; D.4.2 [**Operating Systems**]: Storage Management—*Garbage collection, Allocation / deallocation strategies*; D.4.8 [**Operating Systems**]: Performance—*Measurements*

ISMM'02, June 20-21, 2002, Berlin, Germany.

General Terms

Experimentation, Languages, Measurement, Performance

Keywords

Compilers, garbage collection, Java, memory management, program analysis, liveness analysis

1. INTRODUCTION

GC does not (and in general cannot) collect all the garbage that a program produces. Typically, GC collects objects that are no longer reachable along a path of references starting from a set of *root* references. Additionally, it is well known that liveness information may aid in earlier reclamation of objects, by reducing the set of root references, or by removing dead references (i.e., references that are not subsequently used prior to redefinition) from the heap graph [7]. For example, in [1] liveness information for local reference variables is used to remove dead local reference variables from the root set; thus, some objects could be identified as unreachable and garbage collected earlier.

However, the impact of general liveness on the space collected by GC is as yet unknown. Moreover, the overhead of having GC consult liveness information may be significant. In fact, as noted in [3], it is not clear how to represent heap liveness information in a feasible manner.

In this paper we study the above questions for Java programs. The information is gathered dynamically for a given run for different kinds of liveness information. Thus, these experiments indicate a kind of upper bound on storage savings that could be achieved assuming static liveness information is available.

1.1 Main Results

In this paper we estimate the effect of general liveness information on space consumption of Java applications. We measure the impact of following kinds of liveness information: (i) stack reference liveness (local reference variable liveness in Java), (ii) global reference liveness (static reference variable liveness in Java), (iii) heap reference liveness (instance reference variable liveness or array reference liveness in Java), and (vi) any combination of (i)-(iii).

We instrumented Sun's JDK 1.2 [5] in order to measure (dynamic) liveness information. Experimental results are given for 10 benchmarks, including 5 of the SPECjvm98 benchmark suite. The information for our measurements is gathered dynamically during a program run. Specifically,

we compare the objects reachable from the root set to the ones that are reachable from the root set when ignoring dead references. Thus, these experiments indicate a kind of upper bound on storage savings that could be achieved assuming static liveness information is available.

We measure the impact of liveness assuming two possible interfaces for communicating liveness information to the garbage collector: (a) An idealized interface, where the liveness is recorded for individual heap references, assuming perfect calling context. (b) A more restricted and more natural interface, in which dead heap references are assigned `null`. This interface simulates the way programmers aid GC to reclaim more space. As noted in [1, 9] such an interface may not be practical. However, it provides an estimate for the potential savings expected for a reasonable interface. Designing an actual implementation for these or other interfaces is beyond the scope of this paper.

In a first experiment we consider the idealized interface to GC, which gives an upper bound on any static liveness analysis algorithm combined with any interface to GC. While in some cases this upper bound may turn out to be too loose, we found that for stack reference and global reference liveness analysis, this upper bound suggest small to medium benefits. On average stack reference liveness information buys 2% savings, and combining it with global reference liveness information the savings increases to 9%. Thus, although these liveness schemes suggest known practical solutions both with respect to the static analysis algorithm (at least for stack reference liveness), and with respect to the GC interface, the importance of having these kinds of liveness information is not clear considering their limited impact on space savings.

For heap reference liveness, the idealized interface yields an average 39% potential savings. In this case the question is what part of the potential is achievable considering a "realistic" heap reference liveness analysis and a "reasonable" interface to GC. Using the more restricted but more realistic assign to null interface we measured a potential savings of 15% on average. This leads us to believe that a practical algorithm achieving significant space savings may be possible.

The results of these experiments with the restricted interface also give insights for the kind of information required by heap reference liveness analysis. For example, interprocedural information seems to have major effect on the expected impact. On average we get 6% potential savings assuming only intraprocedural information, and this increases to 15% when interprocedural information is available. In addition, combining heap reference liveness and global reference liveness information using the same restricted GC interface gives negligible additional benefits (less than 0.5% on average) to the potential benefits obtained just by heap liveness. In the future we intend to repeat the latter experiment assuming stack reference liveness information and global reference liveness information is available to better understand the overall impact of a practical liveness framework.

1.2 Related Work

In [1] static stack reference liveness information is used by a type-accurate GC to reduce the set of root references, thus potentially saving some space. In this paper we show a 2% upper bound for space savings achievable through any static stack reference liveness algorithm. This upper bound is close to the actual space savings reported in [1], which lead us to conclude that: (i) the static analysis algorithm reported there is precise enough, and (ii) as noted in [1]: "the main benefit of stack reference liveness analysis is preventing bad surprises".

In [3] the impact of stack reference and global reference liveness information on space consumption is studied for C programs in a garbage-collected environment. Interestingly, the trend of our results for stack reference and global reference liveness information is in line with their results. However, they do not study heap liveness information. In addition, the measurements technique reported there leads to an infeasible amount of information, which requires approximations. In Section 2 we explain why the measurement technique reported in this paper maintains a feasible amount of information, thus no further approximations are required. In particular, our technique is applicable to large applications. Finally, in [3] two runs of the program are required for obtaining the information. In this paper we show a technique to directly compute the impact of complete heap liveness information on space consumption using a single run. This allows us to handle non-deterministic (e.g., multi-threaded) applications.

In our previous work [10, 11], we followed Röjemo and Runciman [8] and measured drag time for objects, which is the time an object is no longer in use; this is the earliest time an object could be collected by any GC. As observed in [8, 11], programmers can use drag information to reduce the space consumption by changing their code. Here we are interested in measuring the impact on space consumption that could be computed by a compiler, in particular, liveness information. Therefore in contrast to drag measurements, in this paper measure the earliest time an object could be collected assuming liveness information were available. This experiment shifts the focus from the use of an object, to the use (and definition) of the references to an object. This shift complicates the task of computing the dynamic information as it demands more trace information.

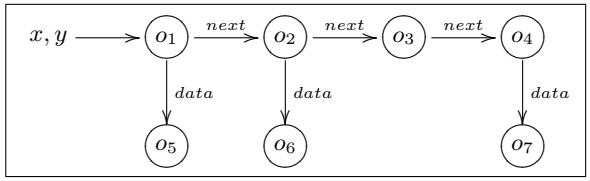
The *drag time* for an object is the time from the last access to the object until the object becomes unreachable. Since the earliest time possible to collect an object is after the last access to it in the run, measuring drag time for objects gives an upper bound for space savings achievable beyond reachability-based garbage collectors. Clearly, if an object is unreachable assuming liveness information is available, then this object is after its last access in the run, i.e., this object is in its drag.

However, an object in its drag could still be reachable even assuming complete liveness information as demonstrated by the code fragment shown in Figure 1(a). Function `foo` processes a singly linked list. Each list element may reference a data object containing a field `f`. We assume the heap at program point `p_1` shown in Figure 1(b). Thus, all `data` references emanating from $o_1 \ldots o_4$ are live, due to their use at `p_2`. Moreover, o_5, o_6, o_7 could not be collected at `p_1` assuming complete liveness information. However, o_5 and o_7 will not be accessed at `p_3` due to the condition at `p_1` (`y.next == null`). Thus, assuming no further accesses to o_5, o_7, these objects are in their drag at `p_1`. Fortunatetly, in Section 4 we show that in practice heap reference liveness information yields potential space savings close to these assuming drag information (on average 39% potential savings vs. 42%, respectively), which lead us to conclude that con-

```
  function foo(L x) {
     L y,t;

     y = x;
     t = null;
p_1: while (y != null) {
p_2:   if (y.data == null && t != null) {
         if (t.data != null) {
p_3:       process(t.data.f);
         }
       }
       t = y;
       y = y.next;
     }
  }
```

(a)



(b)

Figure 1: Drag vs. liveness information example.

centrating on heap reference liveness information is good enough.

1.3 Organization

The remainder of the paper is as follows. Section 2 describes our algorithm for measuring the impact of liveness information and its implementation, assuming an idealized GC interface. Section 3 presents the algorithm and implementation for measuring the impact of liveness in the presence of a restricted GC interface. In Section 4 we discuss the benchmark programs and compare the empirical results of both experiments. We conclude in Section 5.

2. LIVENESS MEASUREMENTS

In this section we present a method for measuring the impact of general liveness schemes on space consumption in a garbage collected environment. Our major observation for approximating the impact of liveness information on space consumption is that there is no need to directly compute dynamic liveness information. In a nutshell, we found that instead it is sufficient to identify the last use of any of the references reaching an object. In Section 2.1 we show that this can be done with a single run of the program by keeping a constant overhead per object (around 60 bytes in our current implementation), and by extending the tracing phase in GC to compute the necessary information[1].

In [3], the impact of stack reference liveness and global reference liveness on space consumption is computed by running the program once to track uses and definitions of references, writing them to a log file. The resulting log file is analyzed in a backward direction to directly compute dynamic liveness information for references, and finally the information is communicated to a second run of the program. Our technique presented here has the following advantages over the above technique. First, it alleviates the problem of the infeasible amount of dynamic liveness information (as reported in [3]). As an immediate result our measurements can handle local variable references, global variable references and heap references without further approximations. Second, the implementation is simpler as there is no need for a mechanism to match a reference in the first run to its corresponding reference in the second run for communicating liveness information. Finally, running the program once allows us to handle non-deterministic (e.g., multi-threaded) applications.

[1]For a non-tracing GC, this may require a separate tracing phase which is specialized to compute just the necessary information for the liveness measurements.

2.1 The Algorithm

Our algorithm computes liveness information while the program (mutator) executes: (i) *when the mutator uses the store*, we record local information for objects, (ii) *when the garbage collector traverses the reachable heap* it propagates global information, and (iii) *when an unreachable object is collected* we compute liveness information.

Table 1 shows the information gathered during the run of the program. As usual, time is measured in bytes allocated so far in the program. All the information is maintained at the level of an object. Columns *property* and *intended meaning* describe the properties we maintain for every allocated object. We classify references as follows: (i) stack references correspond to references on the Java stack, (ii) static references correspond to references in static variables, (iii) other references correspond to references on the native stack, and other special root references, and (iv) heap references correspond to references in instance variables and references in arrays. For each type of reference there are four kinds of properties: (i) liveness (the last time the object was accessed through this reference type), (ii) direct reachability (the last time the object was directly reachable from a variable of this reference type), (iii) path liveness (the last time the object was reachable through a path starting at a live reference of this type), and (iv) reachability (the last time the object was reachable through a path starting at this reference type). Table 1 includes only reference properties needed for computing the liveness impact of stack, static and heap references. For example, $stack_{L^*}(obj)$ keeps the time an object is still reachable along a path starting from a live stack root. Such information is used to determine when *obj* could be collected assuming stack reference liveness information were available. Interestingly, associating the necessary information with objects avoids the need to keep track of all

Property	Intended Meaning	Phase	Event	Update
$heap_L(obj)$	*obj* is referenced by a live heap reference at time $heap_L(obj)$	mutator	$\widehat{use}$ *x.f*	$heap_L(env(x.f)) = Current\ Time$
			$\widehat{use}$ *a[i]*	$heap_L(env(a[i])) = Current\ Time$
$stack_L(obj)$	*obj* is referenced by a live stack root at time $stack_L(obj)$	mutator	*use y*	$stack_L(env(y)) = Current\ Time$
$stack_{L^*}(obj)$	*obj* is reachable along a path starting from a live stack root at time $stack_{L^*}(obj)$	collector	init liveness	$stack_{L^*}(obj) = \max(stack_L(obj), stack_{L^*}(obj))$
			trace children	$stack_{L^*}(obj) = \max\begin{pmatrix} stack_{L^*}(father), \\ stack_{L^*}(obj) \end{pmatrix}$
$stack_R(obj)$	*obj* is referenced by a stack root at time $stack_R(obj)$	collector	trace roots	$stack_R(obj) = Current\ Time$
$stack_{R^*}(obj)$	*obj* is reachable along a path starting from a stack root at time $stack_{R^*}(obj)$	collector	trace children	$stack_{R^*}(obj) = Current\ Time$
$static_L(obj)$	*obj* is referenced by a live static root at time $static_L(obj)$	mutator	*use g*	$static_L(env(g)) = Current\ Time$
$static_{L^*}(obj)$	*obj* is reachable along a path starting from a live static root at time $static_{L^*}(obj)$	collector	init liveness	$static_{L^*}(obj) = \max(static_L(obj), static_{L^*}(obj))$
			trace children	$static_{L^*}(obj) = \max\begin{pmatrix} static_{L^*}(father), \\ static_{L^*}(obj) \end{pmatrix}$
$static_R(obj)$	*obj* is referenced by a static root at time $static_R(obj)$	collector	trace roots	$static_R(obj) = Current\ Time$
$static_{R^*}(obj)$	*obj* is reachable along a path starting from a static root at time $static_{R^*}(obj)$	collector	trace children	$static_{R^*}(obj) = Current\ Time$
$other_R(obj)$	*obj* is referenced by an other root at time $other_R(obj)$	collector	trace roots	$other_R(obj) = Current\ Time$
$other_{R^*}(obj)$	*obj* is reachable along a path starting from an other root at time $other_{R^*}(obj)$	collector	trace children	$other_{R^*}(obj) = Current\ Time$

Table 1: Liveness information gathered during the run. $env(x)$ gives the object referenced by `x`. `y` is a stack variable and `g` is a static variable.

Information	Collection Time
none	$max(stack_{R^*}(obj), static_{R^*}(obj), other_{R^*}(obj))$
heap liveness	$max(heap_L(obj), stack_R(obj), static_R(obj), other_R(obj))$
stack liveness	$max(stack_{L^*}(obj), static_{R^*}(obj), other_{R^*}(obj))$
static liveness	$max(static_{L^*}(obj), stack_{R^*}(obj), other_{R^*}(obj))$
stack + static liveness	$max(stack_{L^*}(obj), static_{L^*}(obj), other_{R^*}(obj))$
heap + stack liveness	$max(heap_L(obj), stack_L(obj), static_R(obj), other_R(obj))$
heap + static liveness	$max(heap_L(obj), static_L(obj), stack_R(obj), other_R(obj))$
heap + stack + static liveness	$max(heap_L(obj), stack_L(obj), static_L(obj), other_R(obj))$

Table 2: Computation of the earliest collection time for an object.

the references in a run and is one of the keys to keeping the amount of information required for the analysis feasible.

The *phase* column indicates which properties are updated during GC, and which properties are updated as result of mutator execution. Specifically, the liveness properties are updated during mutator execution, and the direct reachability, reachability and path liveness properties during GC. A property is updated upon events shown in *event* column. The *update* column shows the new value for the corresponding property when the event occurs. In Section *2.1.1* and Section *2.1.2* we provide further details for property update.

We trigger GC after every 100 KB of allocation in order to propagate the liveness and reachability information at regular intervals.

2.1.1 Actions in the Mutator

When an object reference is used, we update the local information associated with the object. In particular, for a *use r* event, where r is either a heap, stack or a static reference, we set the corresponding liveness property of the object referenced by r, $heap_L$, $stack_L$, or $static_L$ to be the current time. We treat a dereference as two consecutive events. Thus, *use x.f* is split into *use x* and $\widehat{use}$ *x.f*, where $\widehat{use}$ is a special operation that only uses the r-value of `x.f`. Also, *def x.f* is split into *use x* and $\widehat{def}$ *x.f*, where $\widehat{def}$ is a special operation that only uses the l-value of `x.f`. Array reference expressions are handled similarly.

Interestingly, since we associate the liveness properties with the objects, we only need to update the properties for the *use x* and $\widehat{use}$ *x.f* events. These events are sufficient to determine the last time a reference to the object of a particular type (stack, static, or heap) was used. Notice, that after the last use through a particular reference type, the corresponding property will hold the time of last use and the property will not be updated further.

2.1.2 Actions in the Collector

When the collector runs, we update the direct reachability, reachability and path liveness properties of the objects. Specifically, the collector establishes the invariants that these properties correctly describe the status of the heap. For example, when the collection is complete, $stack_{L^*}(obj)$ is set to the last time that *obj* is reachable along a path starting from a live stack root.

Here are the details. We update the path liveness properties for each kind of root. We begin by setting the path liveness properties of each object to the maximum of its current value and the object's liveness property. Next, we propagate path liveness information from each root to its children. In particular, if object A references object B, and the path liveness property of A is greater than that of B, then we set B's value to A's and continue propagating from B. To keep the cost of the propagation proportional to the number of references, we scan stack roots in decreasing order of the path liveness property value of their referenced object. We do the same for static roots. Note that the above implies we visit an object at most twice.

We also update the direct reachability and reachability properties for each kind of root. First we scan the roots and for the objects referenced by the roots, set the directly reachable and reachable properties to the current time according to the kind of the root. For example, if *obj* is referenced by a stack variable, we set $stack_{R^*}(obj)$ to the current time.

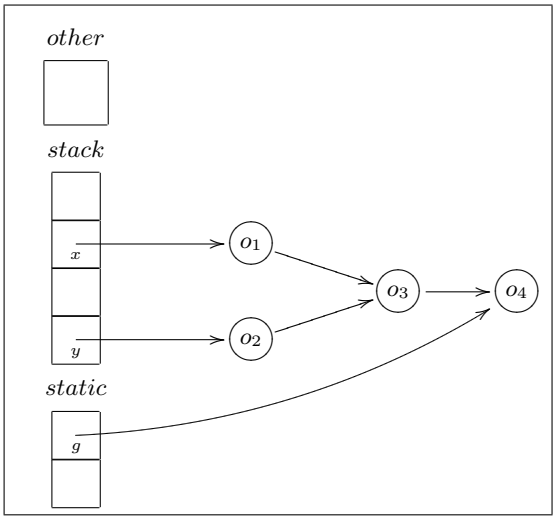


Figure 2: A heap snapshot.

Then, we propagate reachability information from each root to its children. For example, if *obj* is along a path starting from a stack variable, we set $stack_{R^*}(obj)$ to the current time. Notice that the propagation of reachability and liveness information can be combined; thus, in total we visit an object at most twice, scanning the roots once.

2.1.3 Computing Impact of Liveness

When an object is collected, we evaluate the earliest time it could have been collected assuming each of the liveness schemes. Table 2 shows how to compute these times from the object properties. For example, assuming stack reference liveness information were available, an object could be collected when it is no longer reachable along a path from a live stack root, and also not reachable from other roots (e.g., a static root). Figure 2 shows a snapshot of the heap during the run in time t before GC is invoked. If $stack_{L^*}(o_3) < t$, then o_3 could be collected since it is not reachable from other roots. However, o_4 could not be collected at time t since it is still reachable from a static variable (i.e., $static_{R^*}(o_4) \geq t$).

An object's heap liveness property provides information as to whether there is a live reference to it from another object. However, to determine whether the object can be collected, we also need to make sure that it is not directly reachable from a root. For example o_3 could be collected at time t if $heap_L(o_3) < t$. However, o_4 could not be collected at time t since it is still directly reachable from a static variable (i.e., $static_R(o_4) \geq t$).

We compute the average space savings for a particular liveness scheme using the ratio of two reachability integrals. The numerator is the integral for the liveness scheme: we plot the size of the reachable objects assuming the scheme as a function of time and compute the integral under the curve. The denominator is the reachability integral assuming no liveness information.

2.2 Implementation

The instrumented JVM is based on Sun's JVM 1.2 (aka classic JVM). We attach a trailer to every object to keep track of object liveness properties. We do not count the

space taken for this trailer in our data. For debugging purposes our current implementation reports object properties to a log file upon reclamation of the object or upon program termination, and a post processor extract the impact of liveness schemes. We also record for an object its creation time, and its length in bytes. The length includes the handle, the header and the alignment (i.e., the bytes that were skipped in order to allocate the object on an 8 byte boundary), but excludes the trailer.

2.2.1 Updating Information

Object information is updated upon the following events:

Object Creation Creation time and length fields are set.

Reference Use The following events constitute a reference *use*: (1) getting reference field information (e.g., via `getfield` bytecode) updates $heap_L$ property of the referenced object. (2) getting local variable information (e.g., via `aload` bytecode) updates $stack_L$ property of the referenced object. (3) getting global variable information (e.g., via `getstatic` bytecode) updates $static_L$ property of the referenced object.

GC Our implementation follows Section *2.1.2*.

2.2.2 Reporting Information and Impact Computation

To obtain accurate measurements for reachability and liveness, we trigger a full GC every 100 KB.

When an object is freed, we log all of the information collected in its trailer. When the program terminates, we perform a last GC and then we log information for all objects that still remain in the heap.

The rules for the collection of `Class` objects are not the same as for regular objects. Thus, we exclude them and the special objects reachable from them (e.g., *constant pool* strings and per-class security-model objects) from our reports.

Our analyzer reads the log file, and then follows Table 2 to compute the earliest time an object could be collected assuming a particular liveness scheme. Then, the reachability integral for each of the liveness schemes is computed.

3. A FEASIBLE HEAP LIVENESS GC INTERFACE

In this section we consider a feasible interface to communicate heap reference liveness information to GC, in which dead references are assigned `null`. This interface simulates the way programmers aid GC to reclaim more space. Also, it does not require any changes to GC. As noted in [1, 9] such an interface may not be practical. However, we believe that it allows us to estimate the potential savings expected with a reasonable interface.

The algorithm operates in two runs. In the first run we detect the places in the code where assignments to `null` potentially reduce the space, while preserving program semantics, and the program is modified accordingly. Then, the modified program is executed on the same input, in order to evaluate the space savings. Unlike the algorithm in Section 2, this technique is limited to applications with deterministic behavior, due to the second execution of the program.

Event	Action
`p: use x.f`	$SNULL = SNULL \setminus P(lval(x.f))$; $P(lval(x.f)) = p$
`p: def x.f`	$P(lval(x.f)) = p$
`p: use a[i]`	$SNULL = SNULL \setminus P(lval(a[i]))$; $P(lval(a[i])) = p$
`p: def a[i]`	$P(lval(a[i])) = p$

Table 3: Detection of null assignable program points. The set $SNULL$ holds the null assignable program points. For a heap reference h in the run, $P(h)$ holds the last program point that used the l-value of h, i.e. either h itself was used as an r-value, or h was assigned. When the program starts, $SNULL$ is initialized with program points manipulating the heap.

3.1 Algorithm

The idea is to run the program once to identify *null assignable reference expressions*, i.e., a reference expression e at a program point pt, where e could be conservatively assigned null with respect to the current run every time pt is executed. In order to simplify the presentation, we assume the code is normalized so the program point manipulates at most one heap reference expression. The Java bytecode satisfies this requirement. In addition, our algorithm guarantees that e is assigned `null` after pt is executed, only if e is included in the statement in pt. Thus, further on we use the term *null assignable program points*, since it is clear to which expression a `null` value is assigned. Finally, the algorithm could be refined to assign `null` to a heap reference expression occurring in a specific calling context. For example, in our implementation explained in Section 3.2, program points are actually sequences of calling contexts.

At the outset our algorithm assumes that all program points that manipulate the heap are candidates for null assignment. As the program runs, it determines the points where null assignment is impossible, and eliminates them from consideration. At program termination, the remaining points are the null assignable ones. This algorithm is summarized in Table 3.

In particular, the algorithm starts by inserting all candidate program points in $SNULL$. Then it runs the program. Upon a *use x.f* event at program point p, the algorithm concludes that the previous point that used the location denoted by `x.f` cannot be assigned `null`, since it is currently used. Therefore, we remove the previous program point from the set of null assignable program points $SNULL$. In addition, the last program point that used the location denoted by `x.f` (i.e., $P(lval(x.f))$) is set to the current program point p. Upon a *def x.f* event at program point p, we simply set $P(lval(x.f))$ to the current program point p. Usage or a definition of an array reference expression is handled similarly. When the program terminates, $SNULL$ contains the null assignable program points with respect to this run. Finally, the program is modified to assign null to the heap reference expressions in program points included in $SNULL$.

The modified program is executed a second time on the same input and its space consumption is measured. Finally, we compare the space consumption with that in the first run to evaluate the savings.

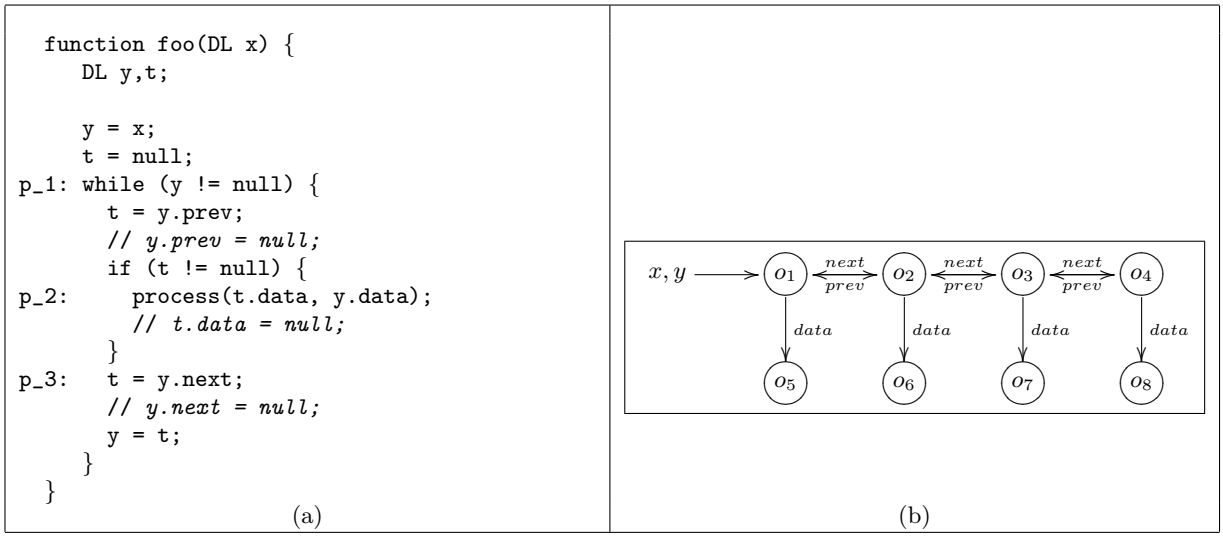


Figure 3: Assign null example.

We demonstrate the algorithm using the code fragment in Figure 3(a) that processes a doubly linked list elements in pairs. We assume the heap at program point `p_1` shown in Figure 3(b). The code is annotated with null assignment for null assignable reference expressions. For example, at `p_3` the expression `y.n` is null assignable. This is since all `next` fields used at `p_3` (i.e., those emanating from $o_1 \ldots o_4$) have no subsequent use in this code fragment. Assuming no further uses of this list in the program, `y.n` is null assignable at `p_3`. Similarly, `t.data` is null assignable at `p_2`. Lastly, `y.data` is not null assignable at `p_2` since the `data` field emanating from $o_1 \ldots o_4$ will be subsequently used in the next iteration through the `t.data` expression.

3.2 Implementation

We instrumented Sun's JVM 1.2 to track uses and definitions of heap references. We attach a trailer to every object to keep track of calling contexts. For every reference instance variable, the trailer contains an entry for the last calling context that used the location of this reference. For feasibility we limit the calling context length to at most k calls. For longer contexts, we only record the suffix of the k last calls (we ran our experiments for $k = 0, 1, 2, 3$.)

Our implementation does not modify the program, but instead the JVM is modified to record the fact that null could be assigned at the proper program points/calling contexts. Since the label of a program point could change across runs, we represent a program point p by the method m containing p, the offset of p in m, and also the opcode in p. Opcode is also used to represent a program point, since our underlying JVM uses byte code rewriting (e.g., it replaces `getfield` bytecode with a `getfield_quick` non-standard bytecode for faster code execution). We also track information about executed calling contexts, and only these contexts are candidates for a `null` assignment. Finally, program points that manipulate references subsequently manipulated by native program points are not candidates for `null` assignment, since a static analysis algorithm is usually not expected to analyze native code.

Object information is updated as described in Table 3 upon the following events:

Heap Reference Use getting reference field information (e.g., via `getfield` or `aaload` bytecode),

Heap Reference Def setting reference field information (e.g., via `putfield` or `aastore` bytecodes)

When the program terminates, we write all null assignable calling contexts to a file. Also, we output the size of the reachable integral computed as explained in Section *2.1.3*.

For the second run, the modified JVM could execute the program and assign null whenever it encounters a null assignable calling context. However, our implementation makes an optimization to allow estimating in a single run the impact of null assignment for several context lengths, and for breaking down the space benefits due to assigning null to field references, and due to assigning null to elements in an array of references, as explained in Section 4. Therefore, in the second run we record for every reference the fact that `null` could be assigned in the trailer of the object containing that reference. Then, every 100KB of allocation we invoke a mark phase per *null assign scheme*, i.e., per a combination of context length and the kind of references being assigned null (e.g., only elements of an array of references). During such a mark phase the information recorded in the object trailer is used to determine whether a reference is considered as assigned null according to the corresponding null assign scheme.

When the program ends, we output the size of the reachable integral per every null assign scheme. The ratio of these integrals and the integral from the first run yields the potential savings. Finally, the output of the original and modified JVM executions of the program are compared in order to provide a sanity check for the correctness of the implementation.

Benchmark	Class	Stmts	Short Description
jess	151	4567	expert system shell
raytrace	25	1479	raytracer of a picture
db	3	512	database simulation
javac	176	12345	java compiler
jack	56	5106	parser generator
mc	15	880	financial simulation
euler	5	726	Euler equations solver
juru	38	2505	web indexing
analyzer	258	35489	mutability analyzer
tvla	218	25264	static analysis framework

Table 4: The benchmark programs.

4. EXPERIMENTAL RESULTS

Our experiments were applied to 10 Java applications shown in Table 4. The second and third columns show the total number of application classes and the total number of application source code statements, respectively. JDK classes and general SPEC classes shared by all SPECjvm98 benchmarks are not included in Table 4. There are 32 general SPEC classes having total of 3173 source code statements.

We employed 5 of the benchmarks from the SPECjvm98 benchmark suite [12]. We did not consider `compress` or `mpegaudio` because they do not use significant amounts of heap memory. `juru` and `analyzer` are internal IBM tools. `euler` and `mc` were taken from Java Grande benchmark suite [4]. The last benchmark, `tvla`, is a three-valued-logic analysis engine framework for static analysis [6].

4.1 The Space Savings due to Liveness

We ran the algorithm described in Section 2 on the benchmarks. Table 5 shows for every benchmark the size of the reachability integral assuming different kinds of liveness information (see Section *2.1.3*). We show information for stack, static, heap liveness information and for every combination of these. Column *w/out liveness* shows the original reachability integral. Column *drag* shows the reachability integral if an object is collected immediately after it is last accessed in the program.

In a similar manner, Table 6 shows for every benchmark the maximum size of the reachable heap in the run for different kinds of liveness information. The integrals indicate an overall view of the space consumption of the program, whereas the maximum heap size is a kind of a feasibility criteria.

Figure 4(a) shows the ratio between the integrals and the original reachability integral, and Figure 4(b) shows the ratio between the maximum heap size and the original maximum heap size. In both figures we also show the average across the benchmarks.

We first discuss the resulting integrals. For stack reference liveness the average potential savings is 2%, and in the best case (*juru*) 12%. In [1] the actual savings obtained by implementing stack reference liveness analysis for Java are shown. The trend of our stack reference liveness results match [1], and also for a simple artificial benchmark (EllisGC) we get duplicate results. Together with [2] we have investigated the differences for other benchmarks, where our dynamic measurements show less potential than the actual savings obtained in [1]. We concluded that our results are not directly comparable due to differences in the experimental environment (i.e., different versions of JDK/JVM, different versions of benchmarks, and different input size used to run the benchmarks). However, the final outcome of both experiments is the same: "the main benefit of stack reference liveness analysis is preventing bad surprises". Another difference in our measurements is that our underlying GC is type-conservative, thus potential savings are shown w.r.t. a conservative GC, where in [1] the base GC is type-accurate. According to [3], this latter difference is not expected to have a major effect on the results.

To the best of our knowledge, the rest of the results are new for object-oriented programs. Moreover, we provide the first study of the effect of heap reference liveness. For static variable liveness, considering the cost of obtaining the information statically, which requires whole program analysis, our experimental results (average of 5% savings) indicate that this optimization may not be profitable. Also for the combination of static variable liveness and stack reference liveness for most of the programs, we get medium savings (average of 9%), while for *juru* and *analyzer* the potential is quite large. This leads us to the same conclusion of "preventing bad surprises".

The comparison of the maximum heap sizes assuming liveness information to the original maximum heap size yield similar results for the benchmarks excluding *juru* and *analyzer*. In *juru* and *analyzer* the profit in maximum heap size is much larger than the overall benefit expressed by the integral size. From our previous experience [11], the reason is that for both benchmarks, stack liveness and/or static variable liveness aid in a few places in the code where a large object, or a group of objects referenced by a single root are kept after their last use.

For heap liveness, our results show an average of 39% potential savings. Moreover, combining heap liveness with other liveness information has negligible effect. This may not come as a surprise, since most of the objects in the heap are referenced solely by heap references. Finally, assuming drag information (i.e., tracking the last access to an object) we get 42% potential savings, which is very close to heap liveness results. Thus, we conclude that heap liveness information potentially brings most of the space benefits achievable beyond reachability-based garbage collectors.

The comparison of the maximum heap sizes assuming liveness information to the original maximum heap size shows that for most of the benchmarks the overall benefit (i.e., the savings in the integral size) is larger (around 7% more savings) than the benefit at the peak (i.e., maximum heap size comparison).

It is interesting to note that our results provide non-trivial upper bounds for potential savings in the integral and the memory footprint between the current GC and an idealized one which has complete liveness information. For example, for *analyzer*, which is a benchmark that allocates memory extensively, the maximum heap size results indicates that in the worst case, the current GC consumes 2.5 times space than the one obtained by an idealized GC.

Figure 5 shows the reachable heap graph over time for *analyzer* considering the following liveness schemes: (i) heap liveness, (ii) stack combined with static variable liveness, and (iii) no liveness information. We see that assuming complete heap liveness the memory footprint of this program re-

benchmark	drag	Liveness Information							w/out liveness
		heap + stack + static	heap + static	heap + stack	heap	stack + static	static	stack	
jess	108.53	118.64	118.81	119.64	119.80	359.84	365.86	391.05	392.97
raytrace	253.73	258.44	258.51	258.96	259.01	640.03	642.99	656.16	656.89
db	497.63	500.57	500.59	500.85	500.87	795.12	796.43	805.27	805.31
javac	1058.19	1065.08	1065.17	1065.90	1065.98	1623.14	1630.31	1640.73	1644.62
jack	86.33	93.33	93.41	93.95	94.03	187.11	194.97	215.99	216.59
analyzer	276.08	284.98	285.15	287.04	287.21	493.91	615.67	657.54	674.39
juru	55.31	58.64	68.54	59.19	69.09	68.01	78.13	77.97	88.25
euler	1936.73	1942.03	1942.06	1946.41	1946.43	2116.45	2125.61	2139.69	2148.82
mc	11423.94	11429.30	11429.32	11433.51	11433.52	11900.48	11901.69	11921.98	11923.18
tvla	318.14	324.95	325.14	332.40	332.58	525.53	528.78	566.08	569.30

Table 5: Reachability integrals (in MB^2) for different liveness kinds.

benchmark	drag	Liveness Information							w/out liveness
		heap + stack + static	heap + static	heap + stack	heap	stack + static	static	stack	
jess	0.61	0.64	0.64	0.65	0.65	1.30	1.33	1.41	1.41
raytrace	2.39	2.42	2.42	2.42	2.42	4.24	4.26	4.34	4.35
db	7.60	7.63	7.63	7.63	7.63	9.50	9.52	9.60	9.60
javac	8.35	8.38	8.38	8.38	8.38	9.16	9.19	9.24	9.25
jack	0.57	0.61	0.61	0.61	0.61	1.28	1.30	1.39	1.39
analyzer	1.38	1.42	1.42	1.43	1.43	2.43	3.40	3.08	3.59
juru	0.46	0.48	0.66	0.49	0.66	0.55	0.82	0.61	0.88
euler	6.72	6.74	6.74	6.75	6.75	7.67	7.70	7.73	7.76
mc	78.93	78.95	78.95	78.96	78.96	82.17	82.20	82.25	82.27
tvla	1.51	1.53	1.53	1.55	1.55	2.27	2.29	2.38	2.40

Table 6: Maximum reachable heap size (in MB) for different liveness kinds.

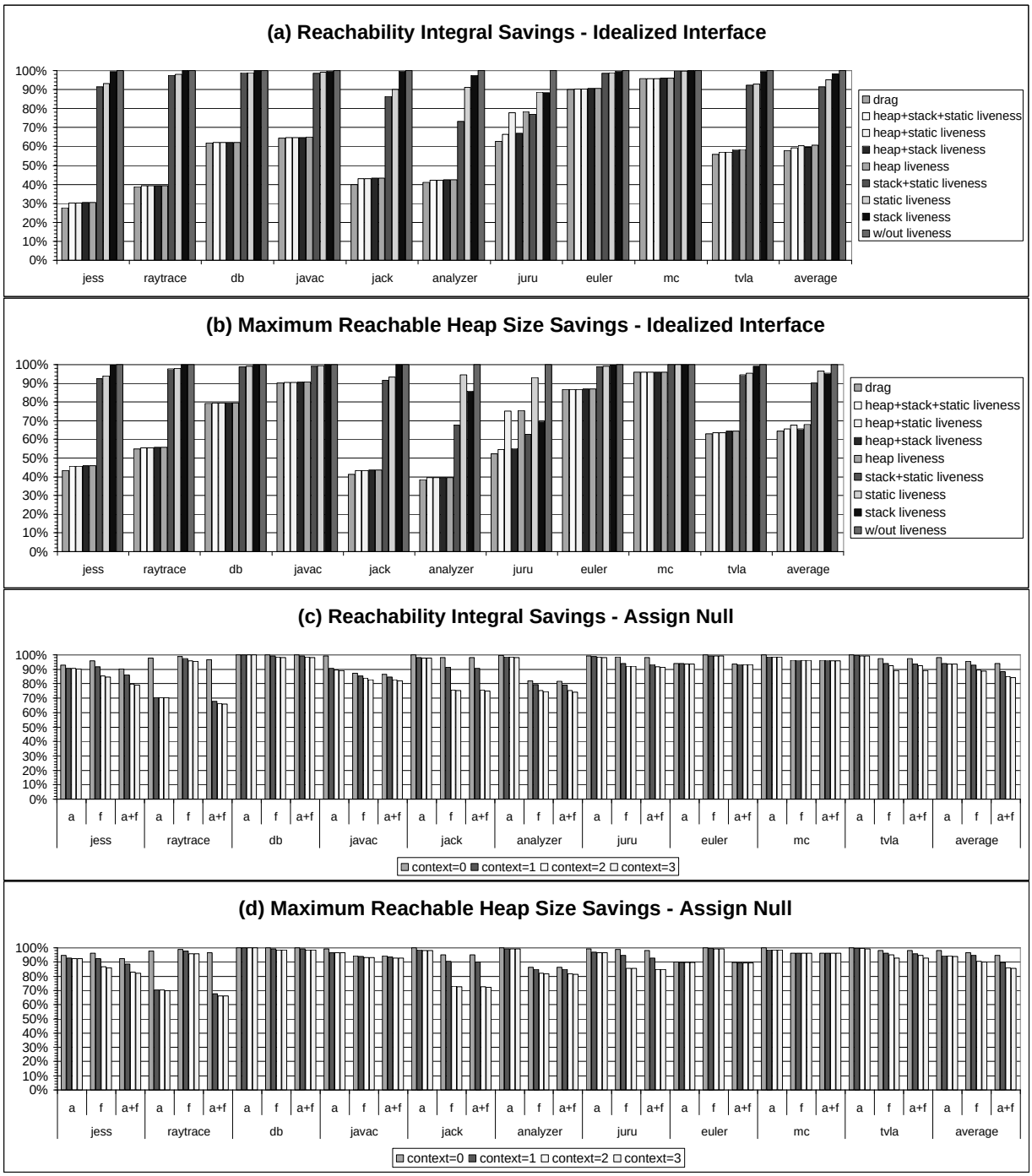


Figure 4: Potential space saving results.

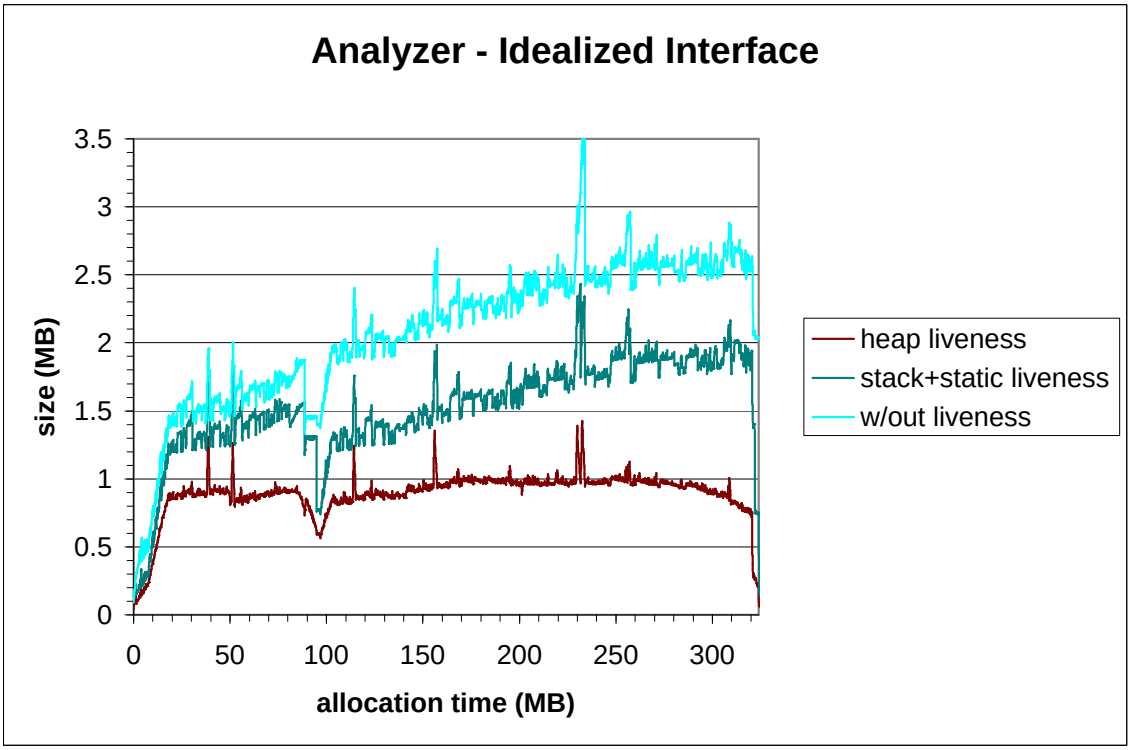


Figure 5: Potential space savings for *Analyzer*.

mains around 920KB. In contrast, assuming complete stack and static variable liveness information, heap consumption increases as the program executes. As expected, the peaks in memory footprint occur simultaneously for these three liveness schemes.

4.2 The Space Savings due to Null Assignments

Figure 4(c) shows the ratio between the integrals of the modified programs assigning `null` to dynamically dead instance variables and the reachability integral in the original program. Similarly, Figure 4(d) shows the ratio for the maximum heap size. We measure potential savings for calling contexts of length 0, 1, 2, and 3. Heap references consist of two kinds, instance fields and elements of an array of references. Since algorithms for approximating the liveness of instance fields may be different in nature than the ones for approximating the liveness of elements in an array of references, we show for each context a breakdown of heap liveness according to these reference kinds. The a column shows potential savings when `null` is assigned only to an element in array of references (i.e., immediately after `aaload` or `aastore`), the f column shows potential savings when `null` is assigned only to an instance field (i.e., immediately after `getfield` or `putfield`), and the $a+f$ column shows overall potential savings when null is assigned to both kinds of heap references.

The average overall space savings for intraprocedural information is 6% while for contexts of length 2, we get 15% potential savings. Breaking down the overall space savings according to heap reference kind, we see that assigning null to instance fields assuming intraprocedural information saves 4.5%, while for contexts of length 2 we get on average 10.5% potential savings. Assigning null to elements in an array of references yields 2% on average assuming intraprocedural information, and 10% potential savings assuming contexts of length 2. In all cases, the added value of contexts of length 3 is insignificant.

On the negative side, we see that this interface provides significantly less potential saving than the one by the idealized interface. For example, in *jack* we get 25% in contrast to the idealized GC in which 57% is saved. However, we believe that the upper bound obtained here is actually more tight than the one reported for the idealized interface since it resembles more closely the effects of static analysis. For example on *db* we get 2% saving here versus 38% with the idealized interface. This is due to the fact that this program accesses a database and thus we believe that the 36% extra saving are due to unused dynamically dead references in the database which cannot be assigned null. Since the input provided in SPECjvm is large enough our algorithm yields that for every point pt, there exists at least one database reference h used in pt which is subsequently used and thus pt is not null assignable.

In another experiment we tried assigning null to static variables. We do not report the detailed results due to space limitations. However, on average, assigning null just to static variables yields less than 1% potential savings, and assigning null both to heap references and static variables yields 15.3% potential space savings, which is less than 0.5% additional benefits comparing to assigning null just to heap references.

benchmark	ref kind	context=0	context=1	context=2	context=3
jess	a	5.38	5.41	5.44	5.44
	f	0.05	0.06	0.06	0.07
	a+f	5.43	5.43	5.45	5.45
raytrace	a	0	0	0	0
	f	0	0	0	0
	a+f	0	0	0	0
javac	a	0.34	0.35	0.41	0.43
	f	0.10	0.13	0.15	0.67
	a+f	0.23	0.26	0.25	0.68
tvla	a	0	0	0	0
	f	0.17	0.23	0.34	0.56
	a+f	0.17	0.22	0.34	0.57

Table 7: The difference (in %) in assign null reachable integral results when considering null assignable program points computed for two runs with different inputs.

4.2.1 Validity of Assign Null Results

Our assign null experiment detects null assignable program points with respect to the current run. However, an optimizer may instrument a program with null assignments, only if a program point is null assignable on *all* execution paths. In order to validate our assign null results, we ran 4 of our benchmarks with another input, and computed null assignable program points. Then, each of these benchmarks was run again with the original input, considering only null assignable program points detected in the run for both inputs.

Table 7 shows the percentage difference in reachable integral potential savings comparing the results of running a benchmark with the original input considering the original set of null assignable program points, and then running the benchmark with the original input considering only the set of program points that are null assignable for both inputs. *ref kind* column denotes the kind of heap reference being assigned null (see Section 4.2). We see that except for the case of assigning null to array elements in *jess* benchmark, the difference is insignificant for all context lengths, and for all reference kinds. This fact gives hope that a precise static analysis algorithm will be able to achieve most of the potential savings shown here.

5. CONCLUSION AND FUTURE WORK

In this paper, we studied the potential space savings due to different kinds of liveness information for Java programs. Our dynamic measurements, which provide an upper bound for space savings achievable by an optimizing compiler show that in general stack reference liveness may yield small benefits, global reference liveness combined with stack reference liveness may yield medium benefits, and heap reference liveness yields the largest potential benefit.

One direction we currently pursue is developing a static analysis algorithm that identifies null assignable program points. Hopefully, such an algorithm will allow obtaining a major part of the potential space savings presented here.

Acknowledgements

We would like to thank David Detlefs and Eliot Moss for their help in validating our empirical results for stack liveness.

6. REFERENCES

[1] O. Agesen, D. Detlefs, and E. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 269–279, June 1998.

[2] D. Detlefs and E. Moss, Feb. 2002. Private Communication.

[3] M. Hirzel, A. Diwan, and A. L. Hosking. On the usefulness of liveness for garbage collection and leak detection. In *ECOOP*, pages 181–206, 2001.

[4] Java Grande Benchmark Suite. Available at http://www.epcc.ed.ac.uk/javagrande.

[5] Sun JDK 1.2. Available at http://java.sun.com/j2se.

[6] T. Lev-Ami and M. Sagiv. TVLA: A framework for kleene based static analysis. In *SAS'00, Static Analysis Symposium*, pages 280–301. Springer, 2000. Available at "http://www.math.tau.ac.il/∼ rumster".

[7] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[8] N. Röjemo and C. Runciman. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In *Proceedings of the 1996 ACM SIGPLAN Int. Conf. on Func. Prog.*, pages 34–41, Philadelphia, Pennsylvania, 24–26 May 1996.

[9] R. Shaham, E. K. Kolodner, and M. Sagiv. Automatic removal of array memory leaks in java. In *Int. Conf. on Comp. Construct.*, pages 50–66, Apr. 2000.

[10] R. Shaham, E. K. Kolodner, and M. Sagiv. On the effectiveness of GC in java. In *Int. Symp. on Memory Management*, pages 12–17. ACM, Oct. 2000.

[11] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient java. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 104–113. ACM, June 2001.

[12] SPECjvm98. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1998. Available at http://www.spec.org/osg/jvm98/.