On the Usefulness of Liveness for Garbage Collection and Leak Detection

Martin Hirzel[1], Amer Diwan[1], and Antony Hosking[2]

[1] University of Colorado
Boulder, CO 80309
{hirzel, diwan}@cs.colorado.edu
[2] Purdue University
West Lafayette, IN 47907
hosking@cs.purdue.edu

Abstract. The effectiveness of garbage collectors and leak detectors in identifying dead objects depends on the "accuracy" of their reachability traversal. Accuracy has two orthogonal dimensions: (i) whether the reachability traversal can distinguish between pointers and non-pointers (type accuracy), and (ii) whether the reachability traversal can identify memory locations that will be dereferenced in the future (liveness accuracy). While prior work has investigated the importance of type accuracy, there has been little work investigating the importance of liveness accuracy for garbage collection or leak detection. This paper presents an experimental study of the importance of liveness on the accuracy of the reachability traversal. We show that while liveness can significantly improve the effectiveness of a garbage collector or leak detector, the simpler liveness schemes are largely ineffective. One must analyze globals using an interprocedural analysis to get significant benefit.[0]

1 Introduction

Garbage collection (GC), or automatic storage reclamation, has many well-known software engineering benefits [29]. First, it eliminates memory management bugs, such as dangling pointers. Second, unlike explicit deallocation, GC does not compromise modularity since modules do not need to know the memory management philosophies of the modules that they use. It is therefore no surprise that even though C and C++ do not mandate GC as part of the language definition, many C and C++ programmers are now using it either for reclaiming memory or for *leak detection*. It is also no surprise that many newer programming languages (e.g., Java [14], Modula-3 [21], SML [20]) require garbage collection. This increased popularity of garbage collection makes it more important than ever to fully understand the tradeoffs between different garbage collection alternatives.

[0] This work was supported by NSF ITR grant CCR-0085792. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

J. Lindskov Knudsen (Ed.): ECOOP 2001, LNCS 2072, pp. 181–206, 2001.

An *ideal* garbage collector or leak detector identifies all heap-allocated objects[1] that are not *dynamically live*. A dynamically-live heap object is one that will be used in the future of the computation. More operationally, a dynamically-live heap object is one that can be reached by following pointers that will be dereferenced in the future of the computation (*dynamically-live pointers*). In order to retain only dynamically-live objects, the ideal garbage collector must be able to exactly identify what memory locations contain dynamically-live pointers. Unfortunately, a real garbage collector or leak detector has no way of knowing what pointers will be dereferenced in the future; thus it may use compiler support to identify an approximation to dynamically-live pointers. The precision of the garbage collector or leak detector in identifying dynamically-live objects depends on the *accuracy* of the compiler support.

There are two dimensions to accuracy: the extent to which the compiler information is able to distinguish pointers from non-pointers (*type accuracy*) and the extent to which the compiler information identifies live pointers (*liveness accuracy*). Prior work [17] has mostly focused only on type accuracy and liveness accuracy has received only a little attention in the literature [1]. In this paper we investigate the effect of different levels of liveness accuracy; in prior work we investigated the effect of different levels of type accuracy [17]. Our approach is to modify a garbage collector (particularly the Boehm-Demers-Weiser collector [7, 9]) to accept and use different combinations of type and liveness accuracy information.

One way to conduct this study is to implement a large number of accuracy schemes in a compiler and garbage collector and to compare their performance. However, accuracy schemes are difficult to implement and thus the above mentioned approach would be infeasible. We therefore take a different approach: we implement the accuracy schemes as a upper-bound approximation in a highly parameterized run-time analysis. This approach is easier since at run time we have perfect alias and control-flow information. However, our approach is limited in that it gives us only an upper bound on the usefulness of accuracy schemes and also requires two identical runs of each program. We do not intend our approach to be used directly for leak detection or garbage collection: the goal of our approach is to collect experimental results that will help to drive subsequent work in leak detection and garbage collection.

To increase the applicability of this study, some of our benchmarks use explicit deallocation while others use garbage collection. Benchmarks in the former group include many C programs from the SPECInt95 benchmark suite. Benchmarks in the latter group include Eiffel programs and some C programs that were designed to be used with a customized or conservative garbage collector.

Our results demonstrate that liveness accuracy significantly improves a garbage collector or leak detector's ability to identify dead objects. However we also find that simple liveness analyses (e.g., intraprocedural analysis of local variables [1]) are largely ineffective for our benchmark programs. In order to get a significant benefit one must use a more aggressive liveness analysis that is interprocedural and can analyze global variables. We also show that our most aggressive liveness analysis is able to identify small leaks in several of our benchmark programs.

[1] We use the term *object* to include any kind of contiguously allocated data record, such as C structs and arrays as well as objects in the sense of object-oriented programming.

The remainder of the paper is organized as follows. Section 2 defines terminology for use in the remainder of the paper. Section 3 further motivates this work. Section 4 reviews prior work in the area. Section 5 describes our experimental methodology and particularly our liveness analysis. Section 6 presents the experimental results. Section 7 discusses the usefulness of our approach in debugging garbage collectors and leak detectors. Section 8 suggests directions for future work and Section 9 concludes.

2 Background

A garbage collector or leak detector identifies unreachable objects using a *reachability traversal* starting from local and global variables of the program.[2] All objects not reached in the reachability traversal are dead and can be freed. In order to identify the greatest number of dead objects, only *live pointers*, that is, pointers that will be dereferenced in the future, must be traversed. Unfortunately, without prior knowledge of the future of the computation it is impossible to precisely identify live pointers. Thus, reachability traversals use conservative approximations to the set of live pointers. In other words, a realistic reachability traversal may treat a non-pointer or a non-live pointer as a live pointer, and may therefore fail to find all the dead objects. The *accuracy* of a reachability traversal refers to its ability to precisely identify live pointers.

There are two dimensions to accuracy: *type accuracy* and *liveness accuracy*. Type accuracy determines whether or not the reachability traversal can distinguish pointers from non-pointers. Liveness accuracy determines whether or not the reachability traversal can identify variables whose value will be dereferenced in the future. Both dimensions require compiler support.

Figure 1 gives an example of the usefulness of type accuracy. Let's suppose the variables *hash* and *ptr* hold the same value (bit pattern) at program point p_3 even though one is a pointer and the other is an integer. If a reachability traversal is not type accurate it will find that the object allocated at p_2 is reachable at point p_5 since *hash* "points to" it. If, instead, the traversal was type accurate, it would not treat *hash* as a pointer and could reclaim the object allocated at p_2 (garbage collection) or report a leak to the programmer (leak detection).

p_1: **int** *hash* = *hashValue*(...);
p_2: **int** *ptr* = (**int**)(*malloc*(...));
p_3: ⟨code using ∗*ptr*⟩
p_4: *ptr* = **null**;
p_5: ...

Fig. 1. Type accuracy example

[2] For simplicity, we do not discuss generational collectors which may also do a reachability traversal starting from selected regions of the heap.

Figure 2 gives an example of the usefulness of liveness accuracy. Let's suppose *parse* returns an abstract syntax tree and that after p_6 *ast* holds the only pointer to the tree. Let's suppose that the variable *ast* is not dereferenced at or after program point p_8 (in other words, it is dead). A reachability traversal that does not use liveness information will not detect that the object returned by *parse* is garbage at program point p_8. On the other hand a reachability traversal that uses liveness information will find that *ast* is dead at program point p_8 and will reclaim the tree returned by *parse* (garbage collection) or report it as a leak to the programmer (leak detection).

p_6: *Tree* $*$*ast* $=$ *parse*$()$;
p_7: *CFG* $*$*cfg* $=$ *translate*$($*ast*$)$;
p_8: $\langle$code that does not use *ast*$\rangle$

Fig. 2. Liveness accuracy example

A major hindrance to both type or liveness accuracy is that they require significant compiler support. In the case of type accuracy the compiler must preserve type information through all the compiler passes and communicate it to the reachability traversal [12]. In the case of liveness accuracy the compiler must conduct a liveness analysis and communicate the liveness information to the reachability traversal. Unlike type information, a compiler does not need to preserve liveness information through its passes if the liveness analysis is the last pass before code generation.

3 Motivation

Prior work has focused almost exclusively on one aspect of accuracy – the ability to distinguish pointers from non-pointers – and has considered liveness only as an afterthought. By separating the two aspects of accuracy, we can identify accuracy strategies that are different from any that have been proposed before and are worth exploring. For example, consider the problem of garbage collecting C programs. Prior work has simply noted that C is unsafe and thus the garbage collector must be conservative (type-inaccurate). While this is true with respect to the pointer/non-pointer dimension of accuracy, it is not true with respect to the liveness dimension. A collector for C and C++ programs which considers all variables with appropriate values to be pointers would improve (both in efficiency and effectiveness) if it knew which variables were live; variables that are not live need not be considered as pointers at GC time even if they appear to be pointers from their value (see example in Figure 2).

Table 1 enumerates a few of the possible variations in each of the two dimensions of accuracy. If prior work has proposed a particular combination of accuracy, the table also references some of the relevant prior work. Many papers have proposed the *no liveness information/full type accuracy* scheme and so we cite only a few of the relevant papers in the table.

Even in this incomplete table, five out of nine combinations are unexplored in the literature. Several of the unexplored combinations have significant potential for advancing

Table 1. Some variations in the two dimensions of garbage collector accuracy

Level of liveness accuracy	Level of type accuracy None	Partial	Full
None	[6]	[4, 10]	[3, 18, 28]
Intraprocedural for local vars			[1, 2, 12, 27]
Interprocedural for local and global vars	(a)	(b)	(c)

the state of the art in leak detection and garbage collection. For example, consider the combination of *interprocedural liveness for local and global variables* with the three possibilities for *pointer information* (marked (a), (b), and (c) in table). Possibility (a) will be useful for unsafe languages, such as C, since it will allow even a type-inaccurate reachability traversal to ignore certain pointers and thus improve both its precision and efficiency. Possibility (c) will improve over the best type-accurate schemes used for type-safe languages such as Java and Modula-3 [1, 12, 27] since it incorporates liveness of globals which we expect to be much more useful than liveness for local variables. Finally, possibility (b) may be useful for either safe or unsafe languages (with some programmer support).

This paper explores a significant part of the accuracy space in order to better understand the different possibilities for liveness and their usefulness in leak detectors and garbage collectors.

4 Related Work

In this section we review prior work on comparing different garbage collection alternatives, type and liveness accuracy for compiled languages, and leak detection.

Shaham *et al.* [23] and Hirzel and Diwan [17] present work that is most relevant to this paper. Shaham *et al.* evaluate a conservative garbage collector using a limit study: They find that the conservative garbage collector is not effective in reclaiming objects in a timely fashion. However, unlike our work, they do not experimentally determine how much of this is due to type inaccuracy versus liveness inaccuracy, or which level of accuracy would make their underlying garbage collector more effective. Hirzel and Diwan [17] present an investigation of different levels of type accuracy using an earlier version of our framework. They demonstrate that the usefulness of type accuracy in reclaiming objects depends on the architecture. In particular, type accuracy is more important for 32-bit architectures than for 64-bit architectures. Hirzel and Diwan investigate only one dimension of accuracy, namely type accuracy, and ignore liveness accuracy in their study.

Bartlett [4], Zorn [32], Smith and Morrisett [24], and Agesen *et al.* [1] compare different garbage collection alternatives with respect to memory consumption. Bartlett [4] describes versions of his mostly-copying garbage collector that differ in stack accuracy. Zorn [32] compares the Boehm-Demers-Weiser collector to a number of explicit memory management implementations. Smith and Morrisett [24] describe a new mostly-copying garbage collector and compare it to the Boehm-Demers-Weiser collector. All

these studies focus on the total heap size. Measuring the total heap size is useful for comparing collectors with the same accuracy, but makes it difficult to tease apart the effects of fragmentation, allocator data structures, and accuracy. Since we are counting bytes in reachable objects instead of total heap size, we are able to look at the effects of garbage collector accuracy in isolation from the other effects. Agesen *et al.* investigate the effect of intraprocedural local variable liveness on the number of reachable bytes after an accurate garbage collection. Besides intraprocedural local-variable liveness we also consider many other kinds of liveness.

Zorn [32], Smith and Morrisett [24], and Hicks *et al.* [16] compare different memory management schemes with respect to their efficiency. Zorn [31] looks at the cache performance of different garbage collectors. We do not look at run-time efficiency but instead concentrate on the effectiveness of garbage collectors in reclaiming objects.

Boehm and Shao [8] describe a technique for obtaining type accuracy for heap objects without compiler support which requires a moderate amount of programmer support. Boehm and Shao do not report any results for the effectiveness of their scheme.

Diwan *et al.* [12], Agesen *et al.* [1], and Stichnoth *et al.* [25] consider how to perform accurate garbage collection in compiled type-safe languages. Diwan *et al.* [12] describe how the compiler and run-time system of Modula-3 can support accurate garbage collection. Agesen *et al.* [1] and Stichnoth *et al.* [25] extend Diwan *et al.*'s work by incorporating liveness into accuracy and allowing garbage collection at *all* points and not just safe points. Even though these papers assume type-safe languages, type accuracy is still difficult to implement especially in the presence of compiler optimizations. Our work identifies what kinds of accuracy are useful for reclaiming objects, which is important for deciding what kinds of accuracy to obtain by compiler analysis. Also, our approach can be used in its current form for identifying leaks in both type-safe and unsafe languages.

Hastings and Joyce [15], Dion and Monier [11], and GreatCircle [13] describe leak detectors based on the Boehm-Demers-Weiser collector [9]. The Boehm-Demers-Weiser collector can also be used as a leak detector [7]. Our scheme uses more accurate information than these detectors and is thus capable of finding more leaks in programs.

5 Methodology

One approach to this study is to implement several different levels of accuracy in a compiler and communicate this information to a reachability traversal. However, because we wanted to experiment with many different levels of accuracy the implementation effort would have been prohibitive since implementing even a single accuracy scheme is a challenging undertaking [12]. We therefore chose a different tactic.

Our basic approach (Figure 3) is to analyze a running program to determine different levels of type and liveness information. This approach is easier than actually building several levels of accuracy since at run time we have perfect aliasing and control flow information. Moreover, at run time we do not have to worry about preserving any information through later optimization passes. An additional advantage is that we can do a direct, detailed, and meaningful comparison between the different memory management schemes. Section 5.1 describes our methodology for collecting type information,

and section 5.2 describes our methodology for collecting different levels of liveness information. Section 5.3 introduces the different accuracy levels that we consider in this paper. Section 5.4 shows how we compare the effectiveness of reachability traversals with different levels of accuracy information. Section 5.5 discusses the limitations of our approach. Section 5.6 describes and gives relevant statistics about our benchmark programs.

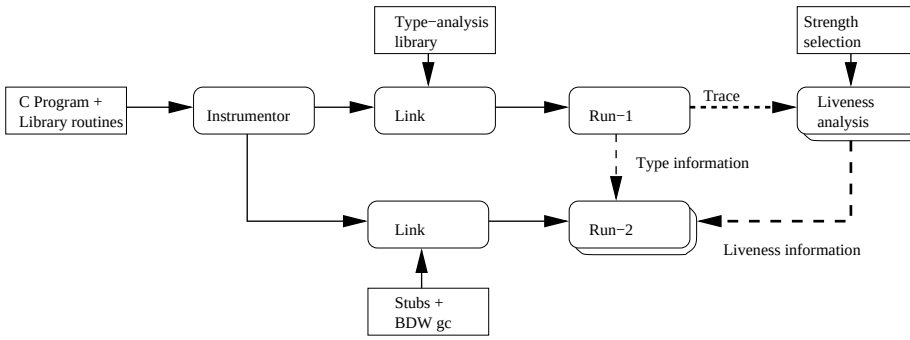

Fig. 3. Framework

5.1 Approach for Type Accuracy

We use the same infrastructure for type accuracy as our previous study on type accuracy [17] so we describe it only briefly here. We convert our C source programs into the SUIF representation [26, 30], instrument the SUIF representation to make calls to a runtime *type-analysis library*, link and run the program (*Run-1*). The type-analysis library precisely tracks the flow of pointers at run time and determines which locations contain pointers. At the end of Run-1, the instrumentation outputs type information in the form of tables that describe which memory locations contain pointers. This information is similar to compiler output in a real garbage collection system for a type-safe language.

Then, we link the same instrumented program with empty stubs instead of the type analysis library and with the Boehm-Demers-Weiser (BDW) garbage collector [7]. We have modified BDW so that it can use the type information during *Run-2*. Since memory addresses of objects may be different in the second run, Run-1 assigns unique identifiers to each heap-allocated object and global variable and uses these identifiers to refer to objects. Run-1 communicates type-accurate information to Run-2 using *location descriptors*, which take one of the following forms:

- $\langle global\_id, offset \rangle$: the global variable identified by *global_id* contains a pointer at *offset*.
- $\langle heap\_id, offset \rangle$: the heap allocated object identified by *heap_id* contains a pointer at *offset*.

– $\langle$*proc_name*, *offset*$\rangle$: activation records for the procedure identified by *proc_name* contain a pointer at *offset*.

We output the above information for every call and allocation point. We do not output any information about pointers in registers since we force all variables to live in memory; registers serve only as scratch space and never contain pointers to objects that are not also reachable from pointers in memory.

The set-up for type accuracy differs slightly from our earlier work on type accuracy [17] in a few aspects. We exclude the activation records of the BDW garbage collector itself from the root set of the reachability traversal. We found and fixed a leak in the BDW collector. Finally, we force the heap to start at a slightly higher address in Run-2 to minimize interference with the data structures needed by our infrastructure.

5.2 Approach for Liveness

Besides generating type information, Run-1 also outputs a trace of events. We analyze this trace to obtain liveness information. In addition to type information, Run-2 can also use the liveness information to improve the precision of its reachability traversals.

Our analysis of the trace mirrors the actions of a traditional backward-flow liveness analysis in a compiler. Like a traditional data-flow liveness analysis, there are two main events in our run-time analysis: uses and definitions. Uses, such as pointer dereferences, make a memory location live at points immediately before the use. Definitions, such as assignments, make the defined memory location dead just before the definition. The run-time analysis is parametrized so that it can realistically simulate a range of static analyses.

Format of the Trace. The trace consists of a sequence of events that are recorded as the program executes. Table 2 describes the kinds of events in a trace. The events in the trace are designed to enable different flavors of liveness analysis.

Some events (such as "assign") refer to memory locations. The trace represents the memory locations using *location descriptor instances* instead of location descriptors as described in Section 5.1, because we need to distinguish between multiple instances of a local variable. Each global location descriptor has only one instance but local location descriptors have multiple instances, one for each invocation of the local variable's enclosing procedure. Each local location descriptor instance, besides identifying its location descriptor, has an attribute, *Home*, which identifies the activation record for which the instance was created. Section 5.2 demonstrates how maintaining location descriptor instances avoids imprecision in analyzing recursive calls.

Basic Algorithm. To obtain liveness information, we perform an analysis on the event trace. In a nutshell, we read the sequence of events in reverse order and keep track of which locations are live at any point during program execution. This approach reflects the fact that liveness of pointers depends on the future, not the past, of the computation.

Our algorithm maintains two data structures: *currentlyLive* and *resultingLiveness*. For each location descriptor instance ℓ, *currentlyLive*(ℓ) indicates whether it is live at

Table 2. Trace events

Event	Example	Description
assign$(lhs, rhs_1, \ldots, rhs_n)$	$x = y + z$	Assignment to location *lhs* from the locations $rhs_1 \ldots rhs_n$. Used to represent normal assignments, parameter passing, and assignment of return value of a call.
use(*rhs*)	$\ldots * x \ldots$	Use of location *rhs*. A pointer dereference is a use. Also passing a parameter to an external function is a use of the parameter.
call()	$\stackrel{\rightarrow}{} f(\ldots)$	Call to a procedure.
return()	$\rightarrow f(\ldots)$	Return from a procedure. (For a longjmp, we generate several *return*-events.)
allocation(p)	*malloc*$(\ldots)$	Allocation of heap object number p (numbered consecutively since program startup).

the current point in the analysis. In other words, as the analysis processes the trace events in reverse order, it keeps track of what is live at any given point of the original execution of the benchmark. The *resultingLiveness* data structure maintains liveness information that will be output at the end of the program. When the liveness analysis finishes, for a stack location descriptor s, *resultingLiveness*$(s) \equiv \{cs_1, \ldots, cs_n\}$ is the set of call sites where s is live, and for a global location descriptor *g*, *resultingLiveness*$(g) \equiv \{p_1, \ldots, p_m\}$ is the set of dynamic calls to malloc where *g* is live (these include the points where we do reachability traversals in Run-2). We use stack location descriptors rather than stack location descriptor instances in *resultingLiveness* to keep the output of the analysis manageable. Note that we output more precise information for globals than for stack variables since maintaining such detailed information for stack locations was infeasible.

As the liveness analysis is processing the trace, it also also tracks the call point at which each active procedure is stopped. For instance, if procedure *p* calls *q*, within the body of *q* the stopping point for the activation record of *p* will be the call to *q* within *p*. Given location description instance x, *HomeCS(*x*)* gives the stopping point of the *Home* activation record of x.

Our analysis never directly reads the *currentlyLive* flags, but instead uses the function *isLive*, which defaults to

$$\textbf{proc}\ \mathit{isLive}(\ell)\ \{\ \textbf{return}\ \mathit{currentlyLive}(\ell);\ \}$$

In Section 5.2, we describe how *isLive* helps to obtain selective liveness.

Table 3 gives the actions that the liveness analysis performs on each event. The actions for *assign* and *use* are similar to the corresponding transfer functions that a compile-time liveness analysis would use. The actions for *call* are, however, more complex, and we motivate and describe them in Section 5.2.

Our algorithm works by keeping the *currentlyLive* flags up-to-date for all locations ℓ. The intuition here is that ℓ must be live prior to any potential dereference of the value it contains; i.e., a *use*, *assign* to another live location, or *call* of an external function that

sees ℓ. When the analysis has completed, it outputs each location descriptor along with its *resultingLiveness*.

Table 3. Liveness analysis

Event	Action
assign$(lhs, rhs_1, \ldots, rhs_n)$	If *isLive*$(lhs) \equiv$ **true**, then make *currentlyLive*$(rhs_1), \ldots,$ *currentlyLive*(rhs_n) true. If none of the rhs_i is the same as the *lhs*, make *currentlyLive* false for *lhs*.
use(rhs)	Make *currentlyLive* true for the location descriptor instance *rhs*.
call$()$	If this is an external call, for each externally visible location ℓ, make *currentlyLive*(ℓ) true. Then, for each stack location descriptor instance s with *isLive*$(s) \equiv$ **true**, add *HomeCS(s)* to the *resultingLiveness* of s's location descriptor.
return$()$	Initialize data structures (such as ones that record the stopping points).
allocation(p)	For each global location ℓ with *isLive*$(\ell) \equiv$ **true**, add the dynamic program point p to the *resultingLiveness* of ℓ.

Analyzing Call Events. To understand the reason for the complexity in analyzing calls, consider the a run of the code segment in Figure 4 where f calls itself recursively just once. Consider the most recent invocation of f (which must be in the *else* branch, since in this example, f recurses just once). The expression $**b$ dereferences the variable c but *from the previous call to* f. Thus, c from the previous invocation of f is live at the recursive call to f. However, even though $**b$ dereferences c, it does not dereference the most recent instance of c and thus, c is not live at the call to g. Calls are the most complex to analyze since that's where we handle such situations precisely.

```
int a;
int **b;
void f(){
    int *c;              /* uninitialized */
    if(...){
        b = &c;
        f();
    else{
        *b = &a;
        g();             /* call site c_g */
        ... **b ...;
    }
}
```

Fig. 4. Recursive call example

The intuition for how we handle calls is as follows. The liveness analysis maintains the *currentlyLive* flags for all location descriptor instances based on the actions in Table 3. When the liveness analysis encounters a call event, it updates the *resultingLiveness* of all stack instances that are live at that call. To update the *resultingLiveness* for a live instance *x*, it adds *HomeCS(x)* to *resultingLiveness(x)*. In other words, call events are the points where we summarize the information in *isLive* into *resultingLiveness*.

Let's consider what happens when we apply our method to the execution of the code in Figure 4. As before, consider a run of the code in where f calls itself recursively just once. Table 4 shows an event trace (in reverse order) of the above program along with the actions our liveness analysis will take. For some events (such as returns) we do not list any actions since these events serve to simply initialize auxiliary data structures. During the trace generation we create two instances of the location descriptor for local variable c: c_1 for the first call to f and c_2 for the second call to f. Note however that our algorithm adds to the *resultingLiveness* of c on behalf of c_1 and not on behalf of c_2. This is correct and precise since c_2 is not dereferenced (or assigned to a variable that is dereferenced) in this run.

Table 4. Processing a trace of the example program

Event	Comment	Analysis action
11: *return*()	outer f returns	
10: *return*()	inner f returns to outer f	
9: *use*(b)	deref of b	*currentlyLive*(b) ← **true**
8: *use*(c_1)	deref of $*b \equiv c_1$	*currentlyLive*(c_1) ← **true**
7: *return*()	g returns to inner f	
6: *call*()	inner f calls g	add *HomeCS*(c_1) to *resultingLiveness*(c)
5: *use*(b)	deref of b	*currentlyLive*(b) ← **true**
4: *assign*(c_1)	assign to $*b \equiv c_1$	*currentlyLive*(c_1) ← **false**
	else-part in inner f	
3: *call*()	outer f calls inner f	no locals live, nothing happens!
2: *assign*(b)	assign to b	*currentlyLive*(b) ← **false**
	then-part in outer f	
1: *call*()	call to outer f	

Selective Liveness. We consider three dimensions that determine the precision of liveness: (i) the region of memory for which we have liveness information (stack, heap, and globals), (ii) whether we compute liveness only for scalar variables or also for record fields and array elements (i.e., *aggregates*), and (iii) whether we compute liveness information intraprocedurally or interprocedurally. We now describe how we vary the above dimensions in the algorithm from Section 5.2.

By changing the implementation of *isLive* we can select the precision level of the first two dimensions. For example, suppose we wish to compute liveness information for scalars in the stack, then we use the implementation of *isLive* in Figure 5. In other

words, for those regions of memory and kinds of variables where we do not want liveness information, we assume they are always live.

```
proc isLive(ℓ){
    if(ℓ ∈ Stack and ℓ ∈ ScalarVars)
        then return currentlyLive(ℓ);
        else return true;
}
```

Fig. 5. *isLive* when computing liveness for scalars in stack

By changing what calls are to external routines we can select the precision of the third dimension. For example, if we wish to mimic intraprocedural analysis then we consider all calls as being to external routines. The action for the *call()*-event in Table 3 will therefore make all externally visible locations (heap locations, global locations, or stack locations whose address gets taken) live at all calls. For interprocedural analysis all calls are to non-external routines. We handle library routines by providing stubs that mimic their behavior.

5.3 Accuracy Levels in This Paper

Table 5. Schemes evaluated

	Area of memory	
	Stack	Stack+Globals
No type accuracy		
None	(N, N)	(N, N)
Intraprocedural scalars	(N, i_s^{scalar})	$(N, i_{sg}^{\text{scalar}})$
Intraprocedural all	(N, i_s^{all})	(N, i_{sg}^{all})
Interprocedural scalars	(N, I_s^{scalar})	$(N, I_{sg}^{\text{scalar}})$
Interprocedural all	(N, I_s^{all})	(N, I_{sg}^{all})
With type accuracy		
None	(T, N)	(T, N)
Intraprocedural scalars	(T, i_s^{scalar})	$(T, i_{sg}^{\text{scalar}})$
Intraprocedural all	(T, i_s^{all})	(T, i_{sg}^{all})
Interprocedural scalars	(T, I_s^{scalar})	$(T, I_{sg}^{\text{scalar}})$
Interprocedural all	(T, I_s^{all})	(T, I_{sg}^{all})

Table 5 gives the schemes that we evaluate in this paper along with abbreviations for the schemes. The first part of the table lists schemes that do not include type accuracy but may include liveness accuracy. The second part of the table lists schemes that

include type accuracy and may also include liveness. The entries in the table are pairs, the first element of which gives the level of type accuracy ($(N, \cdot)$ are schemes with no type accuracy and $(T, \cdot)$ are schemes with type accuracy) and the second element gives the level of liveness accuracy. The "intraprocedural" configurations $(\cdot, i_{\cdot})$ assume the worst case for all externally visible variables (globals and locals whose address has been taken) while the "interprocedural" configurations $(\cdot, I_{\cdot})$ analyze across procedure boundaries for externally visible variables. The "scalars" $(\cdot, \cdot^{\text{scalar}})$ configurations compute liveness information only for scalar variables whereas the "all" $(\cdot, \cdot^{\text{all}})$ configurations compute it for all scalar variables, record fields, and array elements. The "stack" configurations $(\cdot, \cdot_{s})$ compute liveness information only for stack variables whereas the "stack and globals" $(\cdot, \cdot_{sg})$ configurations compute it for locations on the stack and for statically allocated variables. While the abbreviations from Table 5 identify accuracy levels, we will sometimes use them to mean the number of bytes occupied by reachable objects when using that accuracy level.

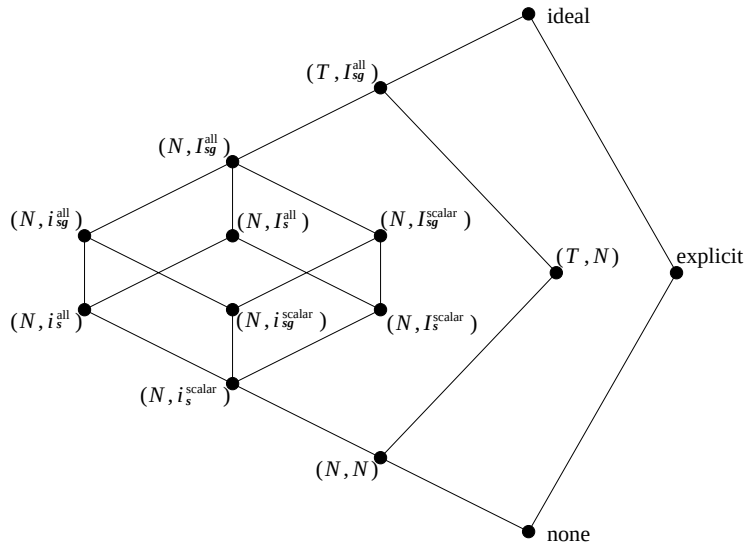

Fig. 6. Memory Management Schemes. Each node in this graph is a memory management scheme. An edge indicates that the scheme with the lower vertical position is strictly weaker than the scheme with the higher vertical position.

Figure 6 presents accuracy schemes organized as a lattice. The order is by strength, with the strongest scheme at the top and the weakest scheme at the bottom.

Note that we decided not to look at liveness for the heap. To see why, let us first imagine what it would mean in our context. Let $\langle \textit{heap\_id}, \textit{offset} \rangle$ be a heap location. If we had heap-accurate liveness for aggregates, we might for example know that even though the heap object *heap_id* contains a pointer at *offset*, that pointer will not be used in the future. But getting this information poses at least two challenges. First, in order to

compute heap liveness we need a precise pointer analysis which is often prohibitively expensive. Second, how to communicate the heap liveness information to the garbage collector? A precise pointer analysis may create many instances of each allocation site and the information may therefore get to be very large. With our trace-based approach, we could of course have obtained heap liveness information, but given the difficulty of obtaining it at compile time, our results would have been a very loose upper bound. Thus, we omitted a study of heap liveness for this paper.

5.4 Measurement Methodology

To collect our numbers, we execute Run-2 multiple times for each benchmark, once for each liveness scheme. To facilitate comparison of the different accuracy schemes, we trigger the reachability traversal at the same time for each level of accuracy. For this study we trigger a reachability traversal every A/n bytes of allocation where A is the total allocation throughout the benchmark run and $n = 50$. Thus for each program and accuracy scheme we end up with a vector of approximately 50 numbers representing the reachable bytes found at each traversal. To compare two liveness schemes, we simply subtract their vectors to determine how they compare at each traversal. The numbers we present in Section 6 are typically averages computed over the difference vectors.

Here is an example for our metric, where for simplicity we assume $n = 3$. Let the conservative garbage collector (N, N) encounter $(100, 200, 200)$ bytes in reachable heap objects after its three collections. Let our strongest liveness scheme (N, I^{all}_{sg}) encounter $(100, 180, 160)$ bytes in reachable heap objects after its three collections. We write $\text{avg}\frac{(N,N)-(N,I^{\text{all}}_{sg})}{(N,N)}$ to mean $\frac{1}{n}\left(\frac{(N,N)_1-(N,I^{\text{all}}_{sg})_1}{(N,N)_1} + \ldots + \frac{(N,N)_n-(N,I^{\text{all}}_{sg})_n}{(N,N)_n}\right)$, which is $\frac{1}{3}\left(\frac{100-100}{100} + \frac{200-180}{200} + \frac{200-160}{200}\right) = 10\%$ in our concrete example. In other words, with strong liveness accuracy, the heap would on average be 10% smaller after garbage collections.

An alternative metric is to measure the heap size (including fragmentation and GC data structures) or the process footprint instead of bytes in reachable heap objects. These are useful metrics but unfortunately not ones we can measure easily in our infrastructure since our instrumentation and extensions to the Boehm-Demers-Weiser collector increase the memory requirements of the host program.

5.5 Limitations

The two main limitations of our approach are: (i) it is a limit study and thus not guaranteed to expose the *realizable* potential of liveness, and (ii) our instrumentation may perturb program behavior and thus, we could suffer from Heisenberg's uncertainty principle.

Our results are an upper bound on the usefulness of liveness information because our analysis has perfect alias information, and because a location may not be live in a particular run, even though there exists a run where it is live. To reduce the possibility of having large errors of this sort, we ran a selection of our benchmarks on multiple inputs and compared the results across the inputs. Section 6.5 presents these results.

Also, we spent significant time manually inspecting the output of our liveness analysis when it yielded a significant benefit. While our manual inspection was not exhaustive (or anywhere close), we found no situations where our liveness analysis' results were specific only to a particular run.

The methodology that we use to obtain our data influences the results itself because we force all local variables to live on the stack, even when they could otherwise have been allocated in registers. Register allocation in a conventional compiler may use its own liveness analysis and may reuse the register assigned to a variable if that variable is dead. Thus, at garbage collection time the dead pointer is not around anymore. In other words, the compiler is passing liveness information to the garbage collector implicitly by modifying the code rather than explicitly. Since register allocators typically use only intraprocedural liveness analysis of scalars, this effect is likely to be strictly weaker than our intraprocedural liveness scheme for scalars on the stack.

5.6 Benchmarks

We used three criteria to select our benchmark programs. First, we picked benchmarks that performed significant heap allocation. Second, we picked benchmarks that we thought would demonstrate the difference between accurate and inaccurate garbage collection. For example, we picked *anagram* since it uses bit vectors which may end up looking like pointers to a conservative garbage collector. Third, we included a number of object-oriented benchmarks.

Table 6 describes our benchmark programs. *Lang.* gives the source language of the benchmark programs. *Lines* gives the number of lines in the source code of the program (including comments and blank lines). *Total alloc.* gives the number of bytes allocated throughout the execution of the program. Two of our benchmarks, *gctest* and *gctest3*, are designed to test garbage collectors [4, 5]. These benchmarks both allocate and create garbage at a rapid rate. The original version of these programs contained explicit calls to the garbage collector. We removed these calls to allow garbage collection to be automatically invoked. The benchmarks *bshift*, *erbt*, *ebignum*, and *gegrep* are Eiffel programs that we translated into C with the GNU Eiffel compiler SmallEiffel. We used the option `-no_gc` and linked the generated C code up with our collector. Likewise, we disabled the garbage collector included in the Lisp interpreter *li* from the SPECInt95 benchmark suite to use our collector instead. The remaining programs use standard C allocation and deallocation to manage memory. We conducted all our experiments on a AMD Athlon workstation.

Due to the prohibitive cost of our analyses,[3] we had to pick relatively short runs for most of the programs. However, for those programs where we were able to do both shorter and longer runs, we found little difference between the two runs as far as our results are concerned.

[3] Some of these benchmarks take over 24 hours on a 850 MHz Athlon with 512MB of memory to run all the configurations.

Table 6. Benchmarks

Name	Lang.	Lines	Total alloc.	Main data structures	Workload
Programs written with gc in mind:					
gctest3	C	85	2 200 004	lists and arrays	loop to 20,000
gctest	C	196	1 123 180	lists and trees	only repeat 5 in listtest2
bshift	Eiffel	350	28 700	dlists	scales 2 through 7
erbt	Eiffel	927	222 300	red-black trees	50 trees with 500 nodes each
ebignum	Eiffel	3 137	109 548	arrays	twice the included test-stub
li	C	7 597	9 030 872	cons cells	`nqueens.lsp`, $n = 7$
gegrep	Eiffel	17 185	106 392	DFAs	`'[A-Za-z]+\-[A-Za-z]+' t`
Programs with explicit deallocation:					
anagram	C	647	259 512	lists and bitfields	`words < input.in`
ks	C	782	7 920	D-arrays and lists	`KL-2.in`
ft	C	2 156	166 832	graphs	`1000 2000`
yacr2	C	3 979	41 380	arrays and structures	`input4.in`
bc	C	7 308	12 382 400	abstract syntax trees	find primes smaller 500
gzip	C	8 163	14 180	Huffman trees	`-d texinfo.tex.gz`
ijpeg	C	31 211	148 664	various image repn.	`testinput.ppm -GO`

6 Results

We now present experimental results to answer the following questions about the usefulness of liveness for garbage collection and leak detection:

1. Does liveness enable us to identify more garbage objects?
2. How does liveness accuracy compare to type accuracy in reclaiming objects?
3. How powerful should a liveness analysis be before it is useful?
4. Do our more powerful liveness schemes allow us to find more memory leaks in our benchmarks?

Sections 6.1, 6.2, 6.3, and 6.4 present results to answer the above questions. Section 6.5 validates our methodology. Section 6.6 discusses the implications of our results for garbage collectors and leak detectors. Finally, Section 6.7 summarizes our results.

6.1 Usefulness of Liveness

In this section we consider whether liveness enables the reachability traversal to detect more of the dead objects as compared to a reachability traversal that does not use liveness information. Table 7 compares our strongest liveness scheme, (N, I^{all}_{sg}), to no liveness, (N, N). To make this and other tables in this paper easier to read, we leave all zero entries blank. Note that there are still some "0" entries in the table: these entries represent values that are less than 1% but not zero.

The first column of Table 7 gives the benchmark program. The second column gives the additional unreachable bytes that (N, I^{all}_{sg}) identifies over (N, N) as a percent of the bytes that (N, N) identifies as reachable. The data in this column is an average

over the data collected at each of the reachability traversals. A non-empty cell in this column means that (N, I_{sg}^{all}) identified more unreachable bytes than (N, N). An empty cell in this column means that (N, N) performed as well as (N, I_{sg}^{all}). The third column gives an indication of the increased memory requirement of (N, N) over (N, I_{sg}^{all}): it compares the maximum number of bytes that are reachable with the two schemes as a percent of the maximum number of bytes that are reachable with (N, N). The fourth column gives the percent of reachability traversals after which (N, I_{sg}^{all}) retained fewer objects than (N, N). Recall that we trigger reachability traversals approximately 50 times for each benchmark run (Section 5.4). A non-empty cell in this column means that at some traversals (N, I_{sg}^{all}) identified more unreachable bytes than (N, N).

Table 7. Usefulness of liveness

Benchmark	avg $\frac{(N,N)-(N,I_{sg}^{\text{all}})}{(N,N)}$%	$\frac{\max(N,N)-\max(N,I_{sg}^{\text{all}})}{\max(N,N)}$%	$\frac{\text{Traversals different}}{\text{Num traversals}}$%
gctest3	0	0	79
gctest			
bshift	42	23	94
erbt	19	6	98
ebignum	13	18	87
li	0		2
gegrep	59	43	98
anagram			
ks			
ft			
yacr2	21	15	90
bc	2	0	98
gzip	11	17	50
ijpeg	1		20

From Table 7 we see that (N, I_{sg}^{all}) benefits 10 out of our 14 benchmark programs. For two of the programs (*gctest3* and *li*) the improvement due to liveness is small. For six of the programs (*bshift*, *erbt*, *ebignum*, *gegrep*, *yacr2*, and *gzip*) liveness reduces the maximum number of reachable bytes by up to 43%. From the fourth column we see that several of the programs leak memory for most of the execution (i.e., the leaks, on average, are not short lived). Thus from these numbers we conclude that liveness (at least in its most aggressive form) has the potential to significantly improve the effectiveness of garbage collectors and leak detectors.

6.2 Liveness versus Type Accuracy

In this section we investigate the individual and cumulative benefits of type and liveness accuracy. Table 8 compares reachability traversals using type accuracy only ((T, N)), liveness accuracy only ((N, I_{sg}^{all})), and both type accuracy and the best liveness accuracy

((T, I_{sg}^{all})). The columns of this table present the difference between the bytes retained by (N, N) and the bytes retained by (T, N), (N, I_{sg}^{all}), and (T, I_{sg}^{all}) as a percent of the bytes retained by (N, N). As with Table 7, the data in Table 8 is an average across all the reachability traversals in a program run. Column 3 of this table is the same as Column 2 of Table 7.

Table 8. Liveness and type accuracy. All benchmarks that see no benefit from liveness or type accuracy are omitted.

Benchmark	avg $\frac{(N,N)-(T,N)}{(N,N)}$%	avg $\frac{(N,N)-(N,I_{sg}^{\text{all}})}{(N,N)}$%	avg $\frac{(N,N)-(T,I_{sg}^{\text{all}})}{(N,N)}$%
gctest3		0	0
bshift		42	42
erbt		19	19
ebignum	0	13	13
li		0	0
gegrep		59	59
yacr2		21	21
bc		2	2
gzip	1	11	12
ijpeg	1	1	1

From Table 8 we see that just adding type information to a reachability traversal yields relatively modest improvements for these benchmark runs (though type accuracy may yield greater benefits on other architectures [17]). In comparison there is a significant benefit to using liveness information in a reachability traversal. From Column 4 we see that there is little benefit to adding type information to liveness for identifying dead objects. In other words, the information that the aggressive liveness analysis computes is sufficient for identifying live pointers. There may, however, be performance benefits to type information since a type-accurate collector can compact reachable memory and thus affect its memory system behavior.

6.3 Strength of Liveness Analysis

In this section we investigate the usefulness of different levels of liveness. Since more precise liveness information is more difficult to implement and expensive to compute, it is important to determine the point of diminishing return for liveness. Table 9 gives the impact of the precision of liveness information on the reachability traversal's ability to identify dead objects. Table 9 is divided into two parts: *Stack liveness* presents the data when we compute liveness only for variables on the stack and *Stack and global liveness* presents the data when we compute liveness for variables on the stack and global variables. Each part has three columns. The first column of each part is the baseline: it shows the benefit of computing simple liveness (i.e., only for scalar variables and using an intraprocedural analysis). We compute the first column of each section in the

same manner as the columns of Table 8. The second and third columns of each section indicate how the value in the first column would increase if we used interprocedural liveness and computed liveness for elements of aggregate variables (i.e., record fields and array elements). There was no benefit to analyzing aggregates in an intraprocedural analysis of stack or global variables and thus we omitted those columns from the table.

Table 9. Varying the strength of the liveness analysis. Columns 2 and 5 (baseline) give the benefit of intraprocedural liveness of scalars for stack and globals. Columns 3, 4, 6, and 7 give the additional benefit of interprocedural analysis and analysis of aggregates over their corresponding baselines. All benchmarks that see benefit from neither liveness nor type accuracy are omitted.

Program	Stack liveness			Stack and global liveness		
	avg $\frac{(N,N)-(N,i_s^{\text{scalar}})}{(N,N)}$%	+IP	+IP+aggr	avg $\frac{(N,N)-(N,i_{sg}^{\text{scalar}})}{(N,N)}$%	+IP	+IP+aggr
gctest3	0			0		
bshift	0			0	3	42
erbt					1	19
ebignum	13	0	0	13	0	0
li					0	0
gegrep	0	9	9	0	23	58
yacr2	0			1	20	20
bc	0			0	1	2
gzip		11	11		11	11
ijpeg			1			1

From Table 9 we see that there is little or no benefit from adding intraprocedural stack liveness for our benchmarks. This is consistent with behavior observed by Agesen *et al.* [1]. Indeed, until we do an interprocedural analysis we get almost no benefit from stack liveness. Note that once we have added interprocedural liveness, analyzing aggregates helps only slightly. Thus, if one is implementing only a stack analysis, then the best bet is to implement an interprocedural liveness analysis and not bother with analyzing non-scalars.

The majority of the benefit of liveness analysis comes from analyzing global variables (see second set of columns in Table 9). The relative importance of local and global variable liveness is not too surprising: unlike local variables, global variables are around for the entire lifetime of the program and thus a dead pointer in a global variable will have a much bigger impact on reachability traversal than a dead pointer in a (relatively short lived) local variable. However, even for global variables, liveness analysis yields little benefit unless the liveness analysis is interprocedural. The cumulative impact of adding aggregate and interprocedural analysis is greater than the sum of the parts. For example, in benchmark *bshift* the benefit of interprocedural analysis is 3% and the benefit of analyzing aggregates is 0%, but the benefit of adding both is 42%.

Figure 7 illustrates how the combined effect of analyzing aggregates and interprocedural analysis is greater than the sum of their parts. In this example s is a global record. Assume for this example that the fields of s are used consistently with their types. If we analyze procedure f using an interprocedural analysis without aggregates then we would have to conclude that the two fields of s may contain pointers at the call to g since the analysis is conservative about record fields. If we analyze procedure f using an intraprocedural liveness analysis that analyzes aggregates then once again we would have to conclude that the fields of s may contain live pointers at the call to g since the intraprocedural analysis assumes the worst case for calls. Only when we analyze procedure f using an interprocedural liveness analysis that analyzes aggregates are we able to determine that the fields of s do not contain pointers.

```
var s : record i : ref int; j : ref int; end
proc f()
    ...
    call g()
    ...
```

Fig. 7. Example of the synergy between analyzing aggregates and doing interprocedural analysis

To summarize, Figure 8 shows both the theoretical (Figure 8(a)) and experimental (Figure 8(b)) relationship between the different liveness analyses. Figure 8(a) is the segment of Figure 6 that contains the liveness accurate memory management schemes. Figure 8(b) is the same graph, but with a different interpretation of vertical position. For each scheme S in (b), the vertical position corresponds to the metric avg$\frac{(N,N)-S}{(N,N)}$%, which is explained in Section 5.4. The horizontal lines in Figure 8(b) connect accuracy schemes that differ in strength only theoretically but not in our experiments.

6.4 Effectiveness in Finding Leaks

The previous sections shed light on the impact of different kinds of liveness information on garbage reclamation or leak detection. In this section we discuss whether or not liveness was able to identify leaks in any of our benchmark programs. We define a *leak* as an object that is never deallocated by the original program but could have been deallocated before the program ended. This is a rather weak notion of leaks, however, since it does not incorporate *timeliness* of deallocation. For example, if an object becomes useless early in the program and is not explicitly deallocated till much later it would not qualify as a leak under our definition.

Of our seven benchmarks that use explicit deallocation (*anagram*, *ks*, *ft*, *yacr2*, *bc*, *gzip*, and *ijpeg*) (N, I_{sg}^{all}) found leaks in *yacr2*, *bc*, and *ijpeg*. Of these, the leaks in *bc* and *ijpeg* are an insignificant percentage of total allocation (less than 1%). The leak in *yacr2* however is significant and accounts for 60% of total allocation (i.e., 60% of of the space is leakage). Since *yacr2* does only a modest amount of total allocation in our run,

a leak of 60% is not as critical as it sounds. However, it is important to keep in mind that most of the benchmarks we used (particularly the C codes) are well-established and well-studied programs; thus it would have been surprising to find significant leaks in them.

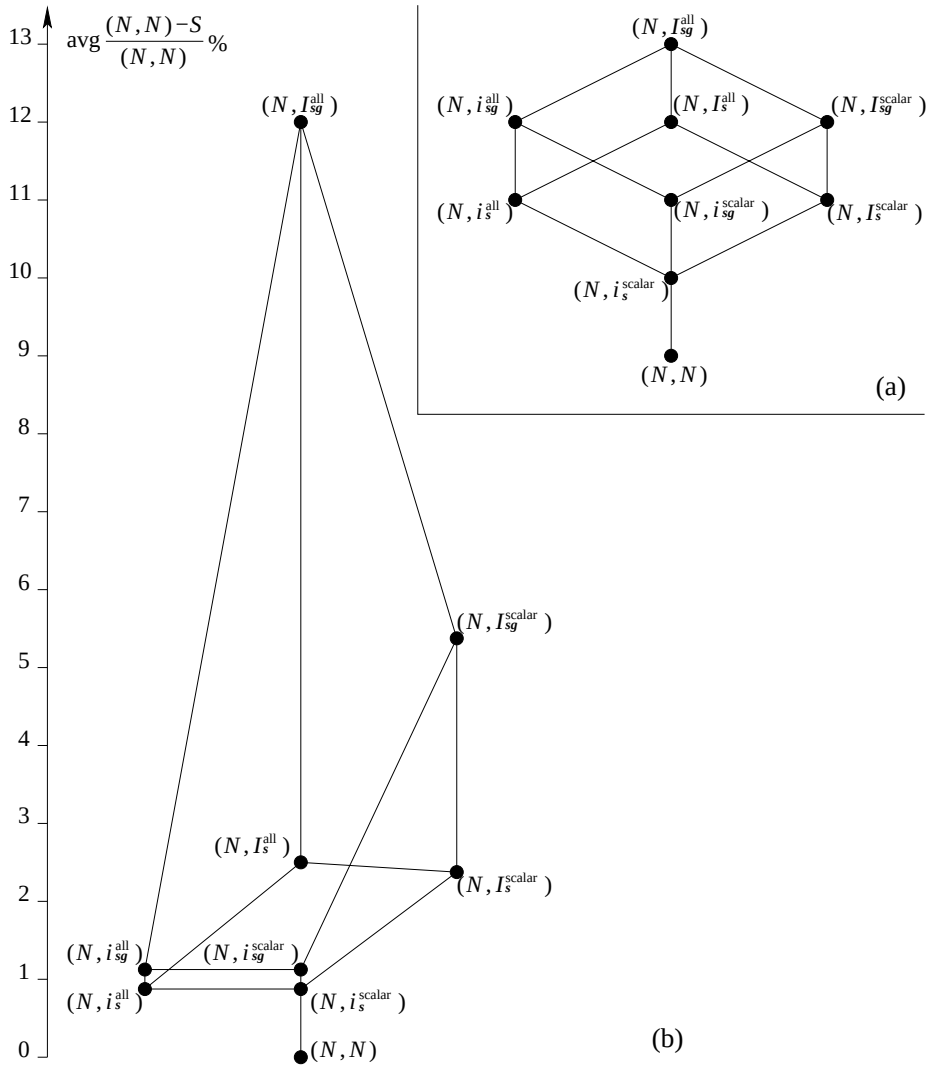

Fig. 8. Theoretical (a) and experimental (b) liveness strength.

6.5 Validation of Our Methodology

Our approach extracts liveness information from a single run of the program and thus it is possible that the liveness information is specific only to that run. In this section we consider how the liveness information varies across runs. A high variation means that our methodology is computing loose upper bounds and thus is severely limited in its usefulness.

To investigate the variation across runs, we ran three benchmarks with a different input and compared the results to our previous runs. If a stack or global location had a different liveness at any point in the two runs we counted that location as "different". Table 10 gives the stack and global locations that are different as a percent of total stack and global locations when using (N, I_{sg}^{all}). The results for other levels of accuracy are similar or better. As with our other tables, we leave the "0" entries blank; 0.0 in this table means that the value is smaller than 0.1% but not 0.

Table 10. Number of stack and global locations that are different as a percent of total static stack and global locations

	Stack		Global	
Benchmark	Count	% different	Count	% different
gegrep	30484	0.7	48717	0.0
yacr2	586	2.7	384	
gzip	2075	1.3	84158	2.2

From Table 10 we see that there is little difference between the liveness information for our two runs. We also measured the effectiveness of different levels of accuracy in identifying dead objects (similarly to Table 7). We found that the results were identical for the two runs in terms of the relative usefulness of the different accuracy schemes. The number of bytes that each liveness scheme was able to identify as dead was of course different between the two runs. Thus, it is likely that our run-time methodology is computing a tight upper bound.

6.6 Implications for Leak Detection and GC

Our results demonstrate that a liveness-accurate reachability traversal will find many more dead bytes than one that is not liveness accurate *even if it is type accurate*. Particularly, even garbage collectors and leak detectors written for unsafe programs can be much more effective with strong liveness information.

A significant advantage of liveness accuracy over type accuracy is that it is more widely applicable since it does not require a compiler to propagate liveness information across its optimization passes and also it does not require type-safe languages. One could even imagine using it to null out pointers in the source code instead of communicating it to the garbage collector in form of tables. Yet the benefits (in reclaiming objects) of liveness information are even greater than the benefits of type information.

Thus, we believe that a liveness analysis deserves to become an integral part of garbage collectors and leak detectors.

6.7 Summary of Results

Our results demonstrate that while liveness accuracy significantly improves a reachability traversal's ability to identify dead objects, the simpler liveness analyses are rarely useful. For liveness accuracy to have a significant impact, the liveness analysis must analyze both local and global variables and use an interprocedural analysis. Adding analysis of aggregate variables further improves interprocedural liveness of local and global variables but has no impact on intraprocedural liveness.

7 Experiences

Besides demonstrating that certain kinds of liveness can be valuable in identifying dead objects, our experiments also had an unexpected side effect: they enabled us to identify leaks in the BDW collector [7]. The BDW collector is a mature and extremely useful tool that has been used heavily by a large user community for over 10 years and there are even commercial leak detection products that are based on this collector [11]. Thus we were surprised to find any leaks in this collector. Our experience leads us to believe that experiments such as ours may be valuable to implementors of garbage collectors and leak detectors in fine tuning their systems.

Broadly speaking there are two kinds of bugs in a garbage collector or leak detector: (i) it can incorrectly identify a live object as dead and (ii) it can fail to identify a dead object. The existence of a bug of the first kind, particularly in a garbage collector, will probably be exposed quickly since freeing a live object will cause the program to exhibit unexpected behavior or to crash. The existence of a bug of the second kind is much harder to detect since it does not cause the program to crash: it just causes the program to use more memory. Since most programmers treat a garbage collector as a black box, they will not realize if the leak is due to a bug in the garbage collector or if it is due to an unfortunate pointer in their own code. All bugs we found in the BDW collector were of the second kind.

How did our experiments help us in finding leaks in the BDW collector? We experimented with a wide range of variations in the BDW collector, some of which minor (such as intraprocedural liveness of local scalar variables) and some of which significant (such as ones involving interprocedural analysis). We discovered the leaks when we saw behavior in one of our variations that did not make sense. For instance, in one case we found that incorporating intraprocedural liveness of global and local variables found many more dead objects than intraprocedural liveness for just local variables. When we tried to imagine how such a situation could happen we ended up with contrived examples which seemed unlikely to appear in real programs. Thus, we investigated further and found the source of the problem: the BDW collector was mistakenly using some of its own global variables as roots. When we provided liveness information for globals to the BDW collector it circumvented BDW's mechanism for finding roots in global variables and thus avoided this problem.

To summarize, garbage collectors and leak detectors are notoriously hard to write and debug. Our experimental methodology provides implementors of these tools with an additional mechanism for identifying potential performance problems.

8 Future Work

Our work demonstrates that while liveness is useful for both garbage collection and leak detection our method is not practical for real-world applications since it requires two identical runs. To remedy this we are working on a compiler support for computing liveness information that obviates the need for two runs at the loss of some precision. We expect that this will not only result in a reachability traversal that users can use for leak detection or garbage collection but it will also allow us to run much larger experiments with liveness. The results in this paper will guide us in determining what kinds of compiler analyses to build in order to improve the effectiveness of reachability traversals.

A limitation of our current infrastructure is that it can handle only C programs or programs that can be converted into C. Given that Java is the current mainstream language that uses garbage collection it would be worthwhile to repeat a similar set of experiments for Java programs. Java programs may behave quite differently from C or Eiffel programs and thus the results may be different for Java programs. We tried using Toba [22] to translate Java programs to C and then use them as benchmarks for this study. Unfortunately the C code that Toba generates even for tiny applications is too large for our infrastructure (since it includes not just the user program but also the Java standard libraries). We are now moving our analysis infrastructure to the Jalapeño JVM [2] which will allow us to experiment with Java programs.

9 Conclusions

We describe a detailed investigation of the impact of liveness and type accuracy on the effectiveness of garbage collectors and leak detectors. By separating the two dimensions of accuracy—*type accuracy* and *liveness accuracy*—we are able to identify interesting new accuracy schemes that have not been investigated in the literature. We use a novel methodology that uses a trace-based analysis to enable us to easily experiment with a wide range of liveness schemes.

Our experiments reveal that liveness can have a significant impact on the ability of a garbage collector or leak detector in identifying dead objects. However, we show that the simple liveness schemes are largely ineffective: we need to use an aggressive liveness scheme that incorporates interprocedural analysis of global variables before we see a significant benefit. Our aggressive liveness schemes are also able to find memory leaks in our suite of well-studied benchmarks.

Acknowledgements

We thank the anonymous reviewers for their helpful comments and suggestions. We also thank Michael Hind and Urs Hoelzle for comments on a draft of this paper, and John DeTreville for fruitful discussions about our methodology and results.

References

[1] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *ACM conference on programming language design and implementation*, pages 269–279, Montreal, Canada, June 1998.

[2] Bowen Alpern *et al.* The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[3] Andrew W. Appel. A Runtime System. *Lisp and Symbolic Computation*, 3(4):343–380, November 1990.

[4] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, CA, February 1988. Also in *Lisp Pointers* 1(6):2-12, April-June 1988.

[5] Joel F. Bartlett. Mostly-copying garbage collection picks up generations and C++. Technical report, DEC Western Research Laboratory, Palo Alto, CA, October 1989.

[6] Hans Boehm, Alan Demers, and Scott Shenker. Mostly parallel garbage collection. In *ACM conference on programming language design and implementation*, pages 157–164, Minneapolis, MN, November 1991.

[7] Hans Boehm, Alan Demers, and Mark Weiser. A garbage collector for C and C++. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[8] Hans Boehm and Zhong Shao. Inferring type maps during garbage collection. In *OOPSLA '93 Workshop on Memory Management and Garbage Collection*, September 1993.

[9] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and experience*, pages 807–820, September 1988.

[10] Dominique Colnet, Philippe Coucaud, and Olivier Zendra. Compiler support to customize the mark and sweep algorithm. In *Proceedings of the International Symposium on Memory Management*, pages 154–165, Vancouver, October 1998.

[11] Jeremy Dion and Louis Monier. Third degree. `http://research.compaq.com/wrl/projects/om/third.html`.

[12] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *ACM conference on programming language design and implementation*, pages 273–282, San Francisco, CA, July 1992.

[13] Great Circle – Real-time error detection and code diagnosis for developers. `http://www.geodesic.com/products/greatcircle.html`.

[14] James Gosling, Bill Joy, and Guy Steele. *The Java language specification*. Addison-Wesley, 1996.

[15] Reed Hastings and Bob Joyce. Fast detection of memory leaks and access errors. In *Proceedings of the Winter '92 USENIX conference*, pages 125–136, 1992.

[16] Michael Hicks, Jonathan Moore, and Scott Nettles. The measured cost of copying garbage collection mechanisms. In *Functional Programming*, pages 292–305, June 1997.

[17] Martin Hirzel and Amer Diwan. On the type accuracy of garbage collection. In *Proceedings of the International Symposium on Memory Management*, pages 1–12, Minneapolis, MN, October 2000.

[18] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. Technical Report 91-47, University of Massachusetts at Amherst, September 1991.

[19] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, 1st edition, 1997.

[20] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[21] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, New Jersey, 1991.

[22] Todd Proebsting, Gregg Townsend, Patrick Bridges, John Hartman, Tim Newsham, and Scott Watterson. Toba: Java for applications – a way ahead of time (WAT) compiler. In *USENIX COOTS*, pages 41–53, June 1997.

[23] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. On the effectiveness of GC in Java. In *Proceedings of the International Symposium on Memory Management*, pages 12–17, Minneapolis, MN, October 2000.

[24] Frederick Smith and Greg Morrisett. Comparing mostly-copying and mark-sweep conservative collection. In *Proceedings of the International Symposium on Memory Management*, pages 68–78, October 1998.

[25] James Stichnoth, Guei-Yuan Lueh, and Michaeł Cierniak. Support for garbage collection at every instruction in a Java compiler. In *ACM conference on programming language design and implementation*, pages 118–127, May 1999.

[26] Stanford University SUIF Research Group. Suif compiler system version 1.x. `http://suif.stanford.edu/suif/suif1/index.html`.

[27] David Tarditi, Greg Morrisett, P. Cheng, C. Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM conference on programming language design and implementation*, pages 181–192, May 1996.

[28] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, 1984.

[29] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *1992 ACM Conference on Lisp and Functional Programming*, pages 32–42, San Francisco, California, June 1992.

[30] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1984.

[31] Benjamin Zorn. The effect of garbage collection on cache performance. Technical Report CU-CS-528-91, University of Colorado at Boulder, May 1991.

[32] Benjamin Zorn. The measured cost of conservative garbage collection. In *Software–Practice and Experience*, pages 733–756, July 1993.