On the Usefulness of Type and Liveness Accuracy for Garbage Collection and Leak Detection

MARTIN HIRZEL, AMER DIWAN, and JOHANNES HENKEL
University of Colorado

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 nonpointers (*type accuracy*), and (ii) whether the reachability traversal can identify memory locations that will be dereferenced in the future (*liveness accuracy*). This article presents an experimental study of the importance of type and liveness accuracy for reachability traversals. We show that liveness accuracy reduces the reachable heap size by up to 62% for our benchmark programs. However, the simpler liveness schemes (e.g., intraprocedural analysis of local variables) are largely ineffective for our benchmark runs: one must analyze global variables using interprocedural analysis to obtain significant benefits. Type accuracy has an insignificant impact on a garbage collector's ability to find unreachable objects in our benchmark runs. We report results for programs written in C, C++, and Eiffel.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*memory management (garbage collection)*

General Terms: Experimentation, Languages, Measurement, Performance

Additional Key Words and Phrases: Conservative garbage collection, leak detection, liveness accuracy, program analysis, type accuracy

Earlier versions of some of the results in this article were presented in Hirzel, M. and Diwan, A. 2000. On the type accuracy of garbage collection. In *Proceedings of the International Symposium on Memory Management (ISMM)* (Oct.), pp. 1–12, and Hirzel, M., Diwan, A., and Hosking, A. 2001. On the usefulness of liveness for garbage collection and leak detection. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (June), pp. 181–206.

This work was supported by the National Science Foundation (NFS) ITR grant CCR-0085792, an NSF Career Award, and a Faculty Partnership Award from IBM.

Any opinions, findings, and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsers.

Authors' address: Department of Computer Science, University of Colorado, Boulder, CO 80309, E-mail: {hirzel;diwan;henkel}@cs.colorado.edu.

1. INTRODUCTION

Garbage collection (GC), or automatic storage reclamation, has many well-known software engineering benefits [Wilson 1992]. It eliminates some memory management bugs, such as dangling pointers. Furthermore, unlike explicit deallocation, GC improves modularity by eliminating memory management philosophies from the interfaces between modules. 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 use it for *reclaiming memory* or for *leak detection*. It is also no surprise that many newer programming languages (e.g., Java [Gosling et al. 1996], Modula-3 [Nelson 1991], SML [Milner et al. 1990]) 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.

An *ideal* garbage collector or leak detector would identify 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 exactly identify what memory locations contain dynamically-live pointers. Unfortunately, a real garbage collector or leak detector can not know what pointers will be dereferenced in the future; thus it may use compiler support to identify an approximation to dynamically-live pointers. The higher the *accuracy* of the compiler support, the fewer objects will the garbage collector or leak detector identify as dynamically live and the more effective will it be at reclaiming dead objects.

There are two dimensions of accuracy: the extent to which the garbage collector can distinguish pointers from non-pointers (*type accuracy*) and the extent to which the garbage collector can identify live pointers (*liveness accuracy*). Although these two dimensions of accuracy are orthogonal, prior work has considered liveness accuracy only as an extension to type accuracy. Since type accuracy is only possible for programs written in type-safe languages, garbage collectors and leak detectors for unsafe languages (such as C and C++) could not yet benefit from liveness accuracy. Moreover, prior work has considered only simple liveness accuracy: liveness of scalar local variables using intraprocedural analysis. In this work, we treat the two dimensions of accuracy orthogonally and explore them both individually and in many combinations. Our work yields valuable insights into how to build and use garbage collectors and leak detectors for both safe and unsafe languages.

We use a novel run-time analysis to conduct this study. The run-time analysis examines a trace of a program execution to extract an optimistic approximation for different levels of liveness and type information. We provide this information to a modified Boehm–Demers–Weiser garbage collector [Boehm et al. 2002], which uses liveness and type accuracy information during garbage collection.

[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.

We show that our approach most likely computes a tight approximation by comparing the results for two different runs of our benchmarks. Our approach allows us to experiment with a wider range of accuracy schemes more easily than the alternative approach of implementing and comparing many accuracy schemes in a compiler. Our approach also allows us to conduct our experiments easily on programs written in many programming languages and executed on many architectures, and thus increases the applicability of our results. We report results for C, C++, and Eiffel programs.

Our results demonstrate that liveness accuracy significantly improves a garbage collector or leak detector's ability to identify garbage objects. Our most accurate liveness scheme, which is interprocedural and analyzes records and arrays, reduces the reachable heap size by up to 62% for our benchmark programs compared to a garbage collector that does not use liveness or type information. We show that our most accurate liveness scheme enables the garbage collector to free objects in a timely fashion compared to explicit deallocation. Type accuracy, on the other hand, does not enable a garbage collector to collect many more objects for our benchmark runs. However, this does not mean that type accuracy is useless for garbage collection; to the contrary, type accuracy is necessary for copying garbage collection. We find that simple liveness analyses (e.g., intraprocedural analysis of local variables [Agesen et al. 1998]) are largely ineffective for our benchmark runs. In order to get a significant benefit one must use a more aggressive liveness analysis that is interprocedural and can analyze global variables. We validate our results using two runs of several benchmark programs.

The remainder of the article is organized as follows: Section 2 defines terminology. Section 3 further motivates this work. Section 4 describes our algorithms for computing type and liveness information. Section 5 describes our experimental methodology. Section 6 presents the experimental results. Section 7 discusses the usefulness of our approach in debugging garbage collectors and leak detectors. Section 8 reviews prior work in the area. Section 9 concludes the article.

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 knowledge of the future of the computation it is impossible to identify live pointers accurately. Thus, reachability traversals use conservative approximations to the set of live pointers. In other words, a realistic reachability traversal may treat a nonpointer or a nonlive pointer as a live pointer, and may therefore fail to find all dead objects. The *accuracy* of a reachability traversal is its ability to accurately identify live pointers.

[2] For simplicity, we only discuss tracing nongenerational collectors.

p_1: **int** *anint* = *random*(...);
p_2: **int** *ptr* = (**int**)(*malloc*(...));
p_3: ⟨code using $*ptr$⟩
p_4: *ptr* = **null**;
p_5: ...

Fig. 1. Type accuracy example.

Table I. Pointers in Three Hardware Platforms

Type	SPARC Solaris	Pentium Linux	Alpha UNIX
void *	0xef5c0aa0	0x08360000	0x0000000140080000
char[]	\239 \ \10 \160	\8 6 \0 \0	\0 \0 \0 \1 @ \8 \0 \0
int	−279180640	137756672	1 1074266112
long	−279180640	137756672	5369233408
float	−6.809955E+28	5.476863E-34	2.652495E-315 2.125

There are two dimensions to accuracy: *type accuracy* and *liveness accuracy*. Type accuracy determines whether or not the reachability traversal can distinguish pointers from nonpointers. 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 illustrates the usefulness of type accuracy. Let us suppose the variables *anint* 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 *anint* "points to" it. If, instead, the traversal was type accurate, it would not treat *anint* as a pointer and could reclaim the object allocated at p_2 (garbage collection) or report a leak to the programmer (leak detection).

Table I describes what pointers look like in three different hardware platforms[3]: *SPARC Enterprise 3500 running Solaris 2*, *Pentium running Linux 2.2 kernel*, and *Alpha running Digital UNIX 4.0D*. For each of these hardware platforms, the table shows the value of the lowest address returned by the allocator for the Boehm–Demers–Weiser collector [Boehm et al. 2002] during a run of a benchmark program (*bc*).[4] The table also shows what that address translates to when interpreted as a string, int, long, or float. In other words, this table shows, for three hardware platforms, the kind of values a string, int, long, or float must have in order for it to be misidentified as a pointer by a conservative garbage collector. That the "int" and "float" rows for Alpha contain two values each because Alpha pointers occupy 64 bits whereas an int or float only requires 32 bits.

From this table, we see that the string interpretation of the pointer yields nonsensical strings for all three hardware platforms. Therefore, we think that it is unlikely that a conservative garbage collector will mistake a text string for a pointer. When pointers are interpreted as integers, we see that on the Alpha

[3] When we refer to hardware platform we mean not just the architecture but also the operating system and the standard libraries on the machine.

[4] These addresses are not the same as the lowest addresses returned by system *malloc*: the BDW collector tries to place objects at high addresses to avoid unnecessary retention due to its conservatism.

p_6: $\mathit{Tree} *\mathit{ast} = \mathit{parse}();$
p_7: $\mathit{CFG} *\mathit{cfg} = \mathit{translate}(\mathit{ast});$
p_8: ⟨code that does not use *ast*⟩

Fig. 2. Liveness accuracy example.

two adjacent integers must have appropriate values in order to be interpreted as a pointer. Thus, it is unlikely that integers will be mistaken for pointers on the Alpha.[5] On the other hand, pointers map to only a single (though large magnitude) integer on the Pentium and SPARC; thus, it is more likely that a conservative garbage collector will retain dead objects on these hardware platforms.

Figure 2 illustrates the usefulness of liveness accuracy. Let us suppose *parse* returns an abstract syntax tree and that after p_6 *ast* holds the only pointer to the tree. Let us 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 data structure 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).

A major hindrance to both type and liveness accuracy is that they require significant compiler support. For type accuracy, compilers must preserve type information through all compiler passes and communicate the type information to the reachability traversal [Diwan et al. 1991]. For liveness accuracy, compilers must conduct a liveness analysis and communicate the liveness information to the reachability traversal. Unlike type information, compilers do not need to preserve liveness information through their passes if they conduct the liveness analysis just before code generation.

3. MOTIVATION

Prior work has focused almost exclusively on one aspect of accuracy—the ability to distinguish pointers from nonpointers—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). Although this is true with respect to the pointer/nonpointer dimension of accuracy, it is not true with respect to the liveness dimension. A collector for C and C++ programs that considers all variables with appropriate values to be pointers would improve (both in efficiency and effectiveness) if it knew which variables are live; variables that are not live need not be considered as pointers at garbage collection time even if they appear to be pointers from their value.

Table II enumerates a few of the possible variations in each of the two dimensions of accuracy. We partition liveness accuracy into three sub-dimensions:

[5] Except if integers are used to implement nonnumeric data like bit vectors.

Table II. Some Combinations of Type and Liveness Accuracy

			Type Accuracy		
Liveness Accuracy			none	partial	full
none			*	**	§
intraprocedural	stack	scalars	(a)		†
		scalars + records			
		scalars + records + arrays			
	stack + globals	scalars			
		scalars + records			
		scalars + records + arrays			
interprocedural	stack	scalars			
		scalars + records			
		scalars + records + arrays			
	stack + globals	scalars			(b)
		scalars + records			
		scalars + records + arrays			

[*] See Boehm et al. [1991].

[**] See Bartlett [1988] and Colnet et al. [1998].

[§] See Appel [1990], Hudson et al. [1991], and Ungar [1984].

[†] See Agesen et al. [1998], Alpern et al. [2000], Diwan et al. [1992], and Tarditi et al. [1996].

(i) whether the analysis is intraprocedural or interprocedural; (ii) whether the analysis computes liveness for stack variables or also for global variables; and (iii) whether the analysis analyzes only scalar variables or also array elements and record fields (aggregates). If prior work has proposed a particular combination of accuracy, the table also references some of the relevant prior work. If many papers have proposed a particular combination, we cite only a few of the relevant papers in the table.

From this table we see that many of the possibilities are unexplored in the literature. Several of the unexplored combinations have significant potential for advancing the state of the art in leak detection and garbage collection. For example, consider the accuracy combination marked (a) which uses weak liveness information but no type information. This scheme could be useful for improving the effectiveness of leak-detectors and garbage collectors for type-unsafe languages such as C and C++. Schemes (a) can also be useful for type-safe languages if we do not need a copying garbage collector. Scheme (b) can be useful for type-safe languages if we want to improve an existing type-accurate collector without adding significant complexity to the compiler.

This article attempts to better understand the usefulness of different kinds of accuracy so that authors of garbage collectors and leak detectors can make more informed decisions.

4. ALGORITHMS FOR LIVENESS AND TYPE INFORMATION

Section 4.1 gives the intuition behind our analyses for computing liveness and type information. Section 4.2 describes the framework in which we conduct our analyses. Sections 4.3 and 4.4 give details of the analyses for type and liveness information, respectively. Section 4.5 discusses the limitations of our approach.

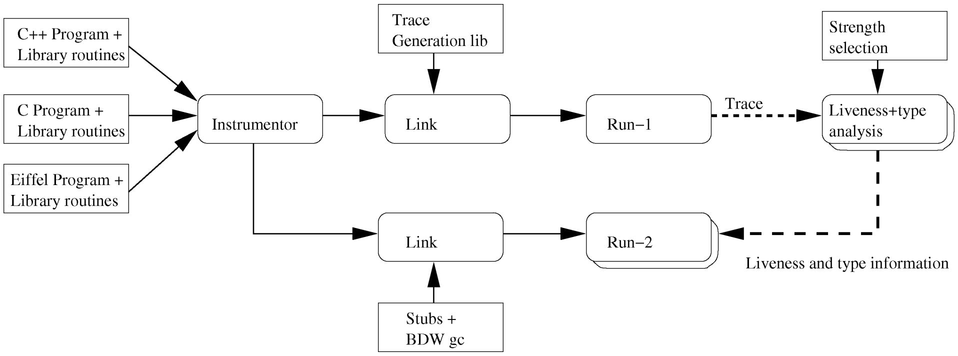

Fig. 3. Framework.

4.1 Intuition Behind Our Approach

To make our study widely applicable and useful, we wanted to explore the impact of a broad range of liveness and type accuracy schemes for programs written in many different styles and languages.[6] Since even a single accuracy scheme is difficult to implement [Diwan et al. 1992], it was clearly infeasible to implement numerous accuracy schemes for several languages. Thus, we use a different approach. Rather than modifying compilers to compute type and liveness information, we analyze traces of program runs to compute the information. Our trace analysis for type information is analogous to a flow and context insensitive type inference in a compiler. Our trace analysis for liveness information is analogous to a flow and context sensitive liveness analysis in a compiler.

Our approach is easier than actually implementing different kinds of accuracy since at run time, when we write our traces, 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. The remainder of Section 4 describes the algorithms in detail and discusses their limitations. Details of the algorithm are not necessary for understanding the rest of the article and thus Sections 4.2, 4.3, and 4.4 may be skipped.

4.2 Framework

Figure 3 describes our experimental framework. We convert C, C++, and Eiffel programs to the SUIF-1 intermediate representation [Stanford University 2002; Wilson et al. 1994]. We then instrument the SUIF representation to make calls to a run-time *trace generation library*, link and run the program (*Run-1*). Run-1 outputs a trace which we analyze to compute and output type and liveness information. Then, we link the same instrumented program with empty stubs instead of the trace generation library and with a modified Boehm–Demers–Weiser (BDW) garbage collector [Boehm et al. 2002]. The garbage collector in *Run-2* uses the liveness and type information from the trace analysis to identify live pointers.

[6] This article presents results for programs written in three languages—C, C++ and Eiffel. Also, our benchmark suite includes programs written both with and without garbage collection.

Table III. Trace Events

Event	Example	Description
assign(lhs,$rhs_1, \ldots, rhs_n$)	$x = y + z$	Assigns a value computed from locations $rhs_1 \ldots rhs_n$ to *lhs*. We use this pattern for assignments, parameter passing, and for returning values from procedures.
addr-assign(*lhs*)	$x = \&y$	Assigns an address to *lhs*. This is really a special case of *assign* that is particularly useful for the type analysis.
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(*call-site*, *callee*)	$^{\rightarrow}f(\ldots)$	Call to a procedure. We use *assign* events to represent parameter passing.
return()	$_{\rightarrow}f(\ldots)$	Return from a procedure. We use *assign* events to represent the return of values. (For a longjmp, we generate several *return*-events.)
allocation(*lhs*)	$p = malloc(\ldots)$	Allocate a heap object and assign pointer to *lhs*.

Table III gives the most important events in a trace. The trace also contains other events that we omit for brevity: for example, the trace also contains events that give information about all local and global variables. The trace makes all assignments explicit: it represents implicit assignments due to parameter passing and return as explicit assignments. Events use *location descriptors* to represent memory locations (e.g., *lhs* in an *assign*). A *location descriptor* uniquely identifies a location in a heap object, global variable, or activation record. For example, if there are two invocations of a procedure containing a variable v, we will create two location descriptors for the variable.

Since registers are not visible at the SUIF level, our trace refers only to memory locations. To correctly handle this limitation, we force all variables to reside in memory; registers serve only as scratch space and never contain pointers to objects that are not also reachable from pointers in memory.

Our methodology assumes that the two runs are identical with respect to variable allocation (heap, global, or stack), have the same object layout, and the same stack layout. To ensure these properties, we use exactly the same binaries for the two runs but link them to different libraries.

4.3 Approach for Type Accuracy

To obtain type information, we analyze the trace in a single forward pass.[7] Table IV describes the actions our analysis takes on each kind of trace event.

In order to make our type analysis realistic and comparable to a type analysis in a compiler, we weaken it in two ways. First, the type analysis uses variables rather than location descriptors for the stack variables. In other words, it does not distinguish between different instances of a stack variable. Second, the type analysis yields flow and context insensitive results. In other words, if a variable contains a pointer at one point in the execution, then we assume that it may contain a pointer whenever the variable is in scope in the execution.

[7] Actually our implementation does this analysis online rather than using the trace.

Table IV. Type Analysis

Event	Action
$assign(lhs, rhs_1, \ldots, rhs_n)$	$isPointer(lhs) \leftarrow isPointer(rhs_1)$ or … or $isPointer(rhs_n)$
$addr\text{-}assign(lhs)$	$isPointer(lhs) \leftarrow$ **true**
$use(rhs)$	ignored
$call(call\text{-}site, callee)$	ignored
$return()$	ignored
$allocation(lhs)$	$isPointer(lhs) \leftarrow$ **true**

The type analysis outputs a table for each call and allocation in the program. The entries in these tables identify stack variables and location descriptors for global and heap locations that contain pointers.

4.4 Approach for Liveness

To obtain liveness information, we analyze the trace in reverse, much like a traditional backward-flow liveness analysis in a compiler. The analysis working backwards reflects the fact that liveness depends on the future, not the past, of the computation.

Like in 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 simulate a range of realistic static analyses.

Our algorithm maintains three data structures: *currentlyLive*, *livenessToOutput*, and *homeCallSite*. For each location descriptor ℓ, the value of $currentlyLive(\ell)$ indicates whether it is live at the current point in the analysis. The data structure *livenessToOutput* collects liveness information that will be output at the end of the program. For a stack variable, we need to know all the static call sites within the enclosing procedure where the variable is live. Thus, for a stack *variable s*, the value $livenessToOutput(s) \equiv \{cs_1, \ldots, cs_n\}$ is the set of static call sites (in the enclosing procedure) where s is live.[8] Analogously to stack variables, for global locations we can keep track of all static call sites in the *program* where the global is live. However, it is more compact to keep track of this information only at the allocation sites. Thus, for a global location descriptor g, the value $livenessToOutput(g) \equiv \{p_1, \ldots, p_m\}$ is the set of dynamic allocation sites where g is live. Note that the information we output for globals is slightly more accurate than that for stack variables because we output dynamic allocation sites for globals and static call sites for stack locations. We do this in order to keep the output of the analysis manageable. For each stack location descriptor, *homeCallSite(x)* gives the return PC for the activation record instance containing x.

[8] This means that we provide the garbage collector only with context-insensitive liveness information for stack variables. While our analysis is context-sensitive and could have produced context sensitive output, we decided against this because no existing garbage collectors use context-sensitive GC tables.

Table V. Liveness Analysis

Event	Action
assign($lhs, rhs_1, \ldots, rhs_n$)	If *isLive*(*lhs*) ≡ **true**, set *currentlyLive*(rhs_1), ..., *currentlyLive*(rhs_n) to **true**. If none of the rhs_i refers to the same location as *lhs*, set *currentlyLive(lhs)* to **false**.
addr-assign(*lhs*)	*currentlyLive*(*lhs*) ← **false**
use(*rhs*)	*currentlyLive*(*rhs*) ← **true**.
call(*call-site*, *callee*)	For external calls, set *currentlyLive*(ℓ) to **true** for each externally visible location descriptor ℓ. Regardless of whether or not the call is to an external routine, for each stack location s with *isLive*(s) ≡ **true** add *homeCallSite(s)* to *livenessToOutput* of the variable corresponding to s.
return()	Initialize data structures.
allocation(lhs)	For each global location g with *isLive*(g) ≡ **true**, add dynamic allocation site to *livenessToOutput*(g). For each stack location s with *isLive*(s) ≡ **true** add *homeCallSite*(s) to *livenessToOutput* of the variable corresponding to s. Set *currentlyLive*(*rhs*) to **false**.

```
        int a;
        int **b;
        void g(){
1:          return;
        }
        void f(){
            int *c;             /* uninitialized */
            if(...){
2:              b = &c;
3:              f();
            else{
4:              *b = &a;
5:              g();
6:              ... **b ...;
            }
7:          return;
        }
```

Fig. 4. Recursive call example.

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 4.4.1, we describe how *isLive* helps to obtain selective liveness.

Table V gives the actions that the liveness analysis performs on each event. The actions for *assign* and *use* are similar to the corresponding transfer functions of a compile-time liveness analysis. The intuition here is that ℓ must be live prior to any potential dereference of the value it contains; that is, a *use*, *assign* to another live location, or *call* of an external function that sees ℓ.

The actions for calls and allocations make sure that the liveness analysis is context sensitive and also update the *livenessToOutput* table. For example, consider 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*

Table VI. Processing a Trace of the Example Program

Event	Comment	(Line)	Analysis action
return()	f returns to *main*	(7)	
return()	f returns to f	(7)	
use(c_1)	deref of $*b \equiv c_1$	(6)	*currentlyLive*(c_1) ← **true**
use(b)	deref of b	(6)	*currentlyLive*(b) ← **true**
return()	g returns to f	(1)	
call(5, g)	f calls g	(5)	add *homeCallSite*(c_1) to *livenessToOutput*(c)
addr-assign(c_1)	assign to $*b \equiv c_1$	(4)	*currentlyLive*(c_1) ← **false**
use(b)	deref of b	(4)	*currentlyLive*(b) ← **true**
call(3, f)	recursive call to f	(3)	no locals live, nothing happens!
addr-assign(b)	assign to b	(2)	*currentlyLive*(b) ← **false**
call($\cdots$, f)	*main* calls f		

```
proc isLive($\ell$){
    if($\ell \in Stack$ and $\ell \in ScalarVars$)
        then return currentlyLive($\ell$);
        else return true;
}
```

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

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.

Let us consider what happens when we apply our method to the execution of the code in Figure 4. Table VI 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 location descriptors for stack variable c: c_1 for the first instantiation of f and c_2 for the second instantiation of f. Note, however, that our algorithm adds to the *livenessToOutput*(c) on behalf of c_1 and not on behalf of c_2. This is correct and accurate since c_2 is not dereferenced (or assigned to a variable that is dereferenced) in this run.

4.4.1 *Selective Liveness.*

We consider three dimensions that determine the accuracy 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., *scalar*, *record*, or record and array (*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 4.4.

By changing the implementation of *isLive* we can select the accuracy level of the first two dimensions. For example, to compute liveness information for scalars in the stack we use the implementation of *isLive* in Figure 5. In other words, we assume those regions of memory and kinds of variables where we do not want liveness information to be always live. When computing liveness information for arrays and records, we treat arrays and records effectively as a collection of scalar variables, with each instance of an array or record giving rise to new instances of their component variables.

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 V 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 nonexternal routines. We handle library routines by providing stubs that mimic their behavior.

4.5 Limitations

The two main limitations of our approach particularly with respect to liveness accuracy 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 influence our results. The remainder of this section discusses these two limitations in detail.

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 two 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 the results of our liveness analysis 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 will be at most as strong as our intraprocedural liveness scheme for scalars on the stack.

5. EXPERIMENTAL METHODOLOGY

Section 5.1 describes the different levels of accuracy we consider in this paper. Section 5.2 describes the metrics we use to measure the usefulness of accuracy. Section 5.3 describes our benchmark programs.

5.1 Accuracy Levels in This Article

Table VII shows the schemes for which we report results in this paper along with abbreviations for the schemes. The entries in the table are pairs, where

Table VII. Schemes Evaluated

			Type Accuracy	
Liveness Accuracy			none	full
none			(N, N)	(T, N)
intraprocedural	stack	scalars	(N, i_s^{scalar})	(T, i_s^{scalar})
		scalars + records	(N, i_s^{record})	(T, i_s^{record})
		scalars + records + arrays	(N, i_s^{aggr})	(T, i_s^{aggr})
	stack + globals	scalars	$(N, i_{sg}^{\text{scalar}})$	$(T, i_{sg}^{\text{scalar}})$
		scalars + records	$(N, i_{sg}^{\text{record}})$	$(T, i_{sg}^{\text{record}})$
		scalars + records + arrays	$(N, i_{sg}^{\text{aggr}})$	$(T, i_{sg}^{\text{aggr}})$
interprocedural	stack	scalars	(N, I_s^{scalar})	(T, I_s^{scalar})
		scalars + records	(N, I_s^{record})	(T, I_s^{record})
		scalars + records + arrays	(N, I_s^{aggr})	(T, I_s^{aggr})
	stack + globals	scalars	$(N, I_{sg}^{\text{scalar}})$	$(T, I_{sg}^{\text{scalar}})$
		scalars + records	$(N, I_{sg}^{\text{record}})$	$(T, I_{sg}^{\text{record}})$
		scalars + records + arrays	$(N, I_{sg}^{\text{aggr}})$	$\left(T, I_{sg}^{\text{aggr}}\right)$

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

In addition to the above configurations, we experimented with partial type accuracy, that is, type information for different regions of memory (stack, heap, globals). We found these configurations to be of negligible value; thus, we omit their results in the remainder of the article.

Note that we do not consider liveness for the heap. To see why, let us imagine what it would mean in our context. If we had accurate liveness information for heap-allocated aggregates, we might, for example, know that even though a heap slot contains a pointer, the slot will not be dereferenced in the future. But getting this information poses at least two challenges. It would be hard to compute heap liveness with an analysis, and it would be hard to use heap liveness in a garbage collector. To compute heap liveness would require a strong pointer analysis, which is often prohibitively expensive. Furthermore, a strong pointer analysis may create many instances of each allocation site and the information may therefore get to be very large. This would be difficult to communicate to, let alone use in, a garbage collector. With our trace-based approach, we could of course have obtained heap liveness information, but given the difficulties

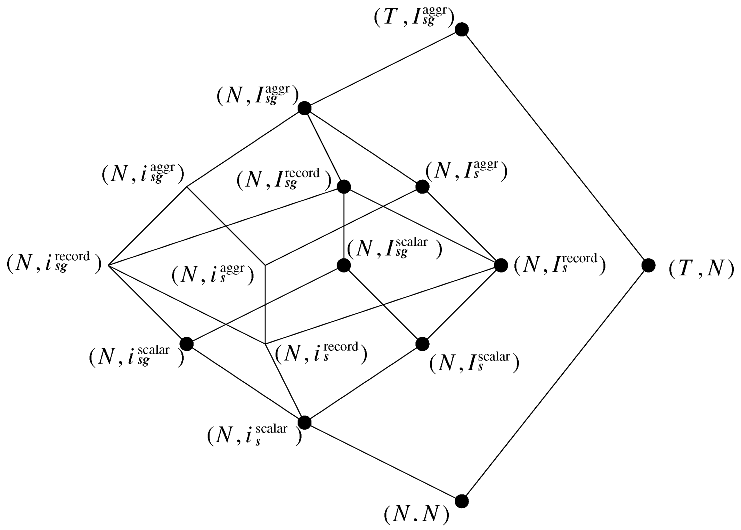
$(T, I_{sg}^{\text{aggr}})$
$(N, I_{sg}^{\text{aggr}})$
$(N, i_{sg}^{\text{aggr}})$
$(N, I_{sg}^{\text{record}})$
(N, I_{s}^{aggr})
$(N, i_{sg}^{\text{record}})$
(N, i_{s}^{aggr})
$(N, I_{sg}^{\text{scalar}})$
$(N, I_{s}^{\text{record}})$
(T, N)
$(N, i_{sg}^{\text{scalar}})$
$(N, i_{s}^{\text{record}})$
$(N, I_{s}^{\text{scalar}})$
$(N, i_{s}^{\text{scalar}})$
$(N.N)$

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.

described above, our results would have been a very loose upper bound. Thus, we omitted a study of heap liveness from this article.

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. To make the figure more readable and because type accuracy offered little benefit in our benchmark programs (Section 6), we omit the different combinations of type accuracy and liveness accuracies $\left(\text{e.g., } \left(T, i_{sg}^{\text{aggr}}\right)\right)$. Although we collected data for all the schemes in Figure 6, we present results for only the schemes that offer some improvement over schemes that are immediately below them in the lattice. These schemes are marked by solid circles. For example, we found that enhancing intraprocedural analysis with record analysis, $\left(\left(N,\ i_{sg}^{\text{record}}\right)\right)$ gave insignificant benefit over intraprocedural analysis of scalars only $\left(\left(N, i_{sg}^{\text{scalar}}\right)\right)$ and thus we do not report further results for $\left(N, i_{sg}^{\text{record}}\right)$.

5.2 Metrics

To conduct our measurements, we execute Run-2 (Figure 3) multiple times for each benchmark, once for each accuracy scheme. We execute all our runs on Pentium-based workstations.[9] To facilitate comparisons between different 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 $n = 50$.[10] Thus, for each program and accuracy scheme, we end up with

[9] In Section 6.4, we see that type accuracy yields different benefits on different architectures.

[10] However, a given benchmark run may have much fewer than 50 reachability traversals if it allocates a number of objects that are larger than A/n bytes.

a vector of approximately 50 numbers representing the reachable bytes found at each traversal. To compare two liveness schemes, we subtract their vectors to determine how they compare at each traversal. We reduce our metric to a single number by reporting the average of the elements of the difference vector.

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 scheme $\left(T, I_{sg}^{\text{aggr}}\right)$ encounter $(100, 180, 160)$ bytes in reachable heap objects after its three collections. We write

$$\text{avg}\frac{(N,N) - \left(T, I_{sg}^{\text{aggr}}\right)}{(N,N)}$$

to mean

$$\frac{1}{n}\left(\frac{(N,N)_1 - \left(T, I_{sg}^{\text{aggr}}\right)_1}{(N,N)_1} + \cdots + \frac{(N,N)_n - \left(T, I_{sg}^{\text{aggr}}\right)_n}{(N,N)_n}\right),$$

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 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.

5.3 Benchmarks

We used three criteria to select our benchmarks. First, we picked programs that perform significant heap allocation. Second, we picked programs 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 picked programs that span a wide variety of programming styles and languages.

Table VIII describes our benchmark programs. *Main data structures* gives the data structures most commonly used in the benchmarks based on code inspection. Table IX gives information about our benchmark programs and their inputs. *Language* gives the source language of the benchmark programs. We have benchmarks in three different languages. *Lines* gives the number of lines in the source code of the program (including comments and blank lines). *Total allocation* gives the number of bytes allocated throughout the execution of the program. *Workload* describes how we run each benchmark. Two of our benchmarks, *gctest* and *gctest3*, are designed to test garbage collectors [Bartlett 1988, 1989]. These benchmarks both allocate objects and create garbage at a rapid rate.

Table VIII. Benchmark Descriptions

Name	Description	Main data structures
Programs using gc:		
gctest3	Synthetic stress test for Bartlett’s collector.	Lists and arrays
gctest	Synthetic stress test for Bartlett’s collector.	Lists and trees
bshift	Measurements on Barrel-shifter topology.	Doubly-linked lists
erbt	Test for red-black-tree package.	Red-black trees
ebignum	Test for arbitrary precision numbers package.	Arrays
li	Lisp interpreter.	Cons cells
gegrep	Pattern finder similar to GNU grep.	DFAs
xerces	XML parser with implementation of Document Object Model.	Document Object Model
Programs using explicit deallocation:		
anagram	Anagram generator.	Lists and bit fields
ks	Kernighan-Schweikert graph partitioner.	Graphs
ft	Finds minimum spanning trees.	Graphs
yacr2	Yet another channel router for circuit layout.	Arrays and structures
bc	GNU bc calculator.	Abstract syntax trees
gzip	GNU gzip compression tool.	Huffman trees
roboop	Robotics simulation package.	Matrices
eon	Probabilistic ray tracer.	Object graph
ijpeg	Image compression and decompression.	Various image repn.

Table IX. Benchmark Statistics

Name	Language	Lines	Total allocation	Workload
gctest3	C	85	2 200 004	loop to 20,000
gctest	C	196	1 123 180	only repeat 5 in listtest2
bshift	Eiffel	350	28 700	scales 2 through 7
erbt	Eiffel	927	222 300	50 trees with 500 nodes each
ebignum	Eiffel	3 137	109 548	twice the included test-stub
li	C	7 597	9 030 872	`nqueens.lsp`, $n = 7$
gegrep	Eiffel	17 185	106 392	`’[A-Za-z]+\-[A-Za-z]+’ t`
xerces	C++	48 452	387 632	Macbeth, act I
anagram	C	647	259 512	`words < input.in`
ks	C	782	7 920	`KL-2.in`
ft	C	2 156	166 832	`1000 2000`
yacr2	C	3 979	41 380	`input4.in`
bc	C	7 308	12 382 400	find primes smaller 500
gzip	C	8 163	14 180	`-d texinfo.tex.gz`
roboop	C++	11 806	587 544	10
eon	C++	30 115	54 540	Kajiya, 10x10
ijpeg	C	31 211	148 664	`testinput.ppm -GO`

The benchmarks *bshift*, *erbt*, *ebignum*, and *gegrep* are Eiffel programs that we translated into C using the GNU Eiffel compiler SmallEiffel. The benchmarks *xerces*, *roboop*, and *eon* are C++ programs that we translated into C using EDG [Edison Design Group 2002]. We made minor modifications to *xerces* and *eon* to get them to work with our infrastructure. Many of our benchmarks (including all Eiffel programs) were written with garbage collection in mind and some even included a garbage collector as part of the program (e.g., *li* and *xerces*). We modified these programs to use our garbage collector instead of their own.

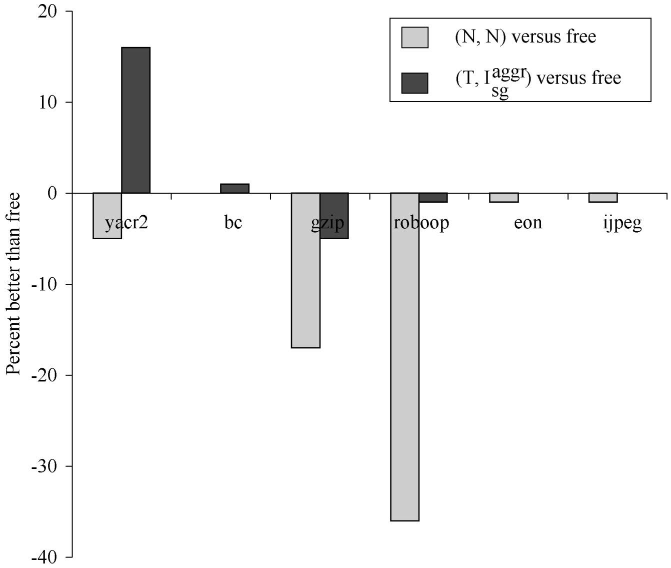

Fig. 7. Accurate and conservative garbage collection versus free. We present results for benchmarks that use explicit deallocation and for which there is a difference between explicit deallocation and (N, N).

Due to the prohibitive cost of our analyses,[11] 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 (Section 6.5).

6. RESULTS

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

(1) How does garbage collection compare to explicit deallocation? (Section 6.1)
(2) Does accuracy improve the effectiveness of garbage collection? (Section 6.2)
(3) How much accuracy does a garbage collector need in order to reclaim the most objects? (Section 6.3)
(4) Does the benefit of type accuracy depend on the underlying architecture and memory layout? (Section 6.4)
(5) Does our methodology yield valid results? (Section 6.5)

6.1 GC Versus Explicit Deallocation

Figure 7 compares the performance of our most inaccurate ((N, N)) and accurate $\left((T, I_{sg}^{\text{aggr}})\right)$ collectors to explicit deallocation. The height of the (N, N) *versus free* bar presents the average difference between the bytes retained by

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

(N, N) and explicit deallocation as a percentage of the bytes retained by (N, N). The height of the $\left(T, I_{sg}^{aggr}\right)$ *versus free* bar presents the average difference between the bytes retained by $\left(T, I_{sg}^{\text{aggr}}\right)$ and explicit deallocation as a percentage of the bytes retained by (N, N). A negative bar in Figure 7 indicates that garbage collection frees fewer bytes than explicit deallocation. A positive bar means that garbage collection frees more bytes than explicit deallocation.

Figure 7 gives the data only for benchmarks that use explicit deallocation. Also, since (N, N), $\left(T, I_{sg}^{\text{aggr}}\right)$, and explicit deallocation collect exactly the same number of objects for *anagram*, *ks*, and *ft*, we omit these programs.

From this figure, we see that explicit deallocation is better than conservative garbage collection ((N, N)) for five of the nine benchmarks that use explicit deallocation. The performance of $\left(T, I_{sg}^{\text{aggr}}\right)$ is more impressive: it finds leaks in at least two benchmark programs (*yacr2* and *bc*). In the other programs, $\left(T, I_{sg}^{\text{aggr}}\right)$ is still close in performance to explicit deallocation. Thus, as far as reclaiming memory or detecting leaks is concerned, $\left(T, I_{sg}^{\text{aggr}}\right)$ compares favorably to explicit deallocation while (N, N) is much worse than explicit deallocation.

6.2 Usefulness of Accuracy

Section 6.2.1 compares the ability of type accurate, liveness accurate, and conservative collectors in reclaiming objects. Section 6.2.2 compares how much work each kind of collector needs to do at garbage collection time.

6.2.1 *Usefulness of Liveness and Type Accuracy for Reclaiming Objects.*

In this section, we investigate the individual and cumulative benefits of type and liveness accuracy. Figure 8 compares reachability traversals using type accuracy only ((T, N)), liveness accuracy only $\left(\left(N, I_{sg}^{\text{aggr}}\right)\right)$, and both type accuracy and the best liveness accuracy $\left(\left(T, I_{sg}^{\text{aggr}}\right)\right)$. The bars of this graph present the difference between the bytes retained by (N, N) and the bytes retained by (T, N), $\left(N, I_{sg}^{\text{aggr}}\right)$, and $\left(T, I_{sg}^{\text{aggr}}\right)$ as a percentage of the bytes retained by (N, N). As with Figure 7, the data in Figure 8 is an average across all the reachability traversals in a program run.

From Figure 8, we see that just adding type information to a reachability traversal yields modest improvements for only three programs (*gzip*, *roboop*, and *ijpeg*). In comparison, there is a significant benefit to using liveness information in a reachability traversal. We also see that there is no benefit to adding type information to liveness for identifying garbage objects. In other words, the information that the aggressive liveness analysis computes is sufficient for identifying live pointers in our benchmark runs. Note, however, that type information is necessary in environments that use copying garbage collection.

From Figure 8, we also see that, for six out of the seventeen benchmarks (*gctest3*, *gctest*, *li*, *anagram*, *ks*, *ft*), there is no benefit from any kind of accuracy at least for our runs. (Some of the subsequent graphs will omit these benchmarks in order to make them more readable.)

While Figure 8 shows that accuracy enables a garbage collector to collect more bytes on average, it does not say whether the leaks in a conservative

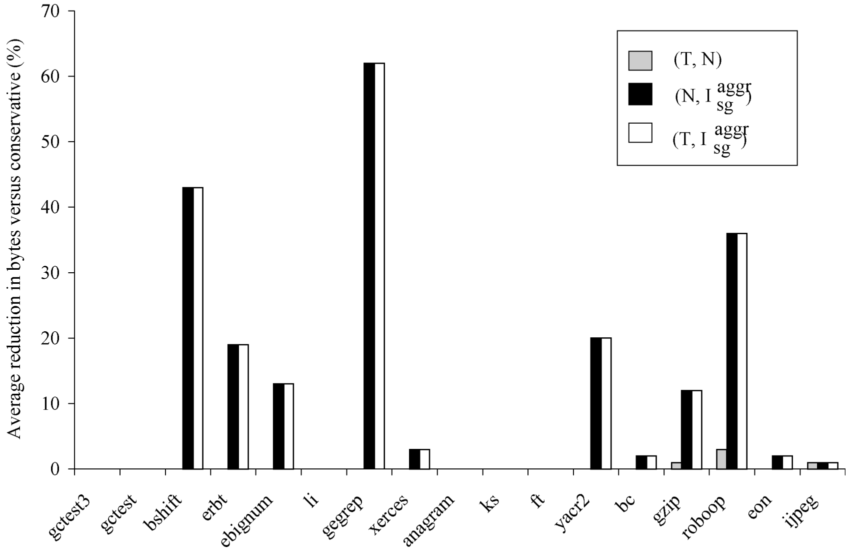

Fig. 8. Benefits of liveness and type accuracy.

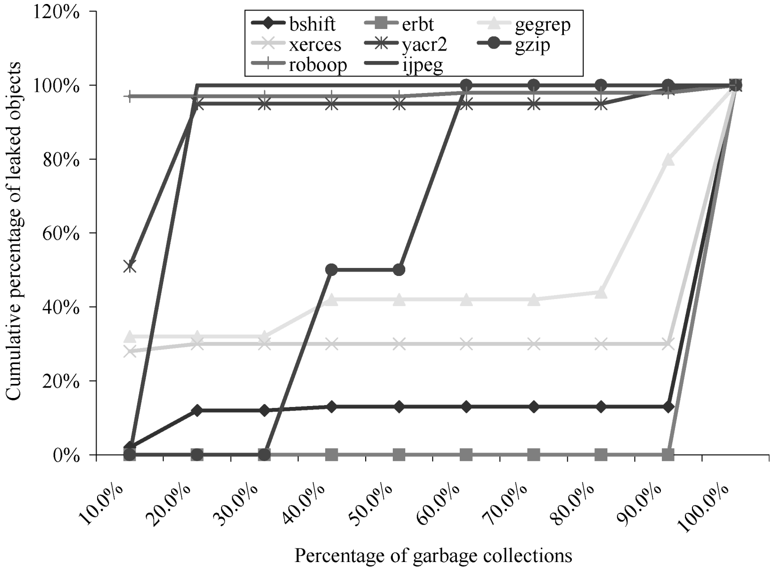

Fig. 9. For how long are objects leaked?

collector are short lived (e.g., (N, N) collects exactly the same objects as $\left(T, I_{sg}^{\text{aggr}}\right)$ but one garbage collection later) or long lived (e.g., there are some objects that $\left(T, I_{sg}^{\text{aggr}}\right)$ frees that (N, N) never identifies as unreachable).

Figure 9 addresses the nature of the improvement that $\left(T, I_{sg}^{\text{aggr}}\right)$ provides over (N, N) for those of our programs that benefit the most from accuracy. A point (x, y) in Figure 9 says that y% of the leaked objects are leaked for x% or fewer garbage collections. For example, in benchmark *gzip* 50% of the objects

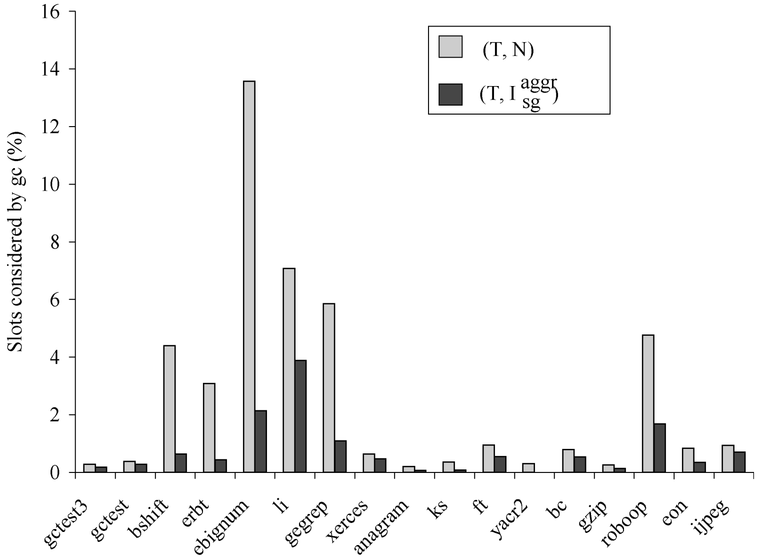

Fig. 10. Percentage of stack slots that the garbage collector must examine.

leaked by (N, N) $\left(\text{in relation to } \left(T, I_{sg}^{\text{aggr}}\right)\right)$ are leaked for up to 40% of the garbage collections (recall that we trigger garbage collection approximately 50 times for each benchmark program). Curves that remain low until they get to the far right of the graph (e.g., *bshift*) indicate benchmarks where (N, N) leaks objects for a significant portion of overall program execution. Such leaks are likely to cause real memory problems in long-running programs.

6.2.2 *Usefulness of Liveness and Type Accuracy for Reducing Effort.* Besides enabling a garbage collector or leak detector to identify more garbage objects, accuracy can also reduce the amount of work for root (i.e., global and stack) processing. A conservative collector must look at all stack and global locations to find pointers while an accurate collector has tables that allow it to skip examining many locations that are not live or not pointers. While accurate collection will most likely inspect fewer locations, it will incur the overhead of decoding the tables; we ignore table decoding overhead in this study. Figures 10, 11, and 12 give the percentage of stack, global, and heap locations that garbage collectors using (T, N) and $\left(T, I_{sg}^{\text{aggr}}\right)$ would have to examine; in contrast, a conservative collector must examine all stack and global locations.

From these figures, we see that even though type accuracy does not reclaim many more objects than conservative collection, it does significantly reduce the work for the garbage collector. Liveness accuracy further reduces the number of memory slots that a garbage collector or leak detector must examine. The benefit of liveness accuracy is most prominent in the stack (Figure 10).

The benefit of accuracy in the heap (Figure 12) comes from two sources. First, as with stack and global variables, accuracy allows a garbage collector to skip nonpointer or nonlive slots. Second, since accurate garbage collectors free up more objects than conservative collectors, there are fewer slots in the heap with

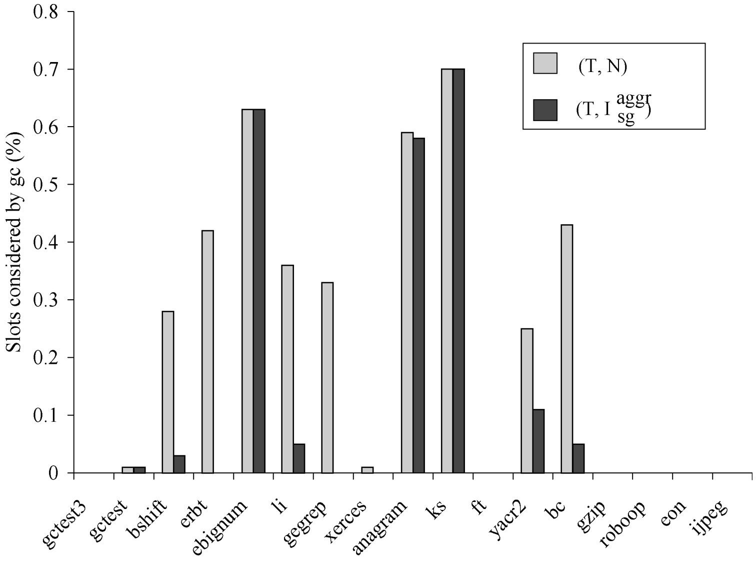

Fig. 11. Percentage of global slots that the garbage collector must examine.

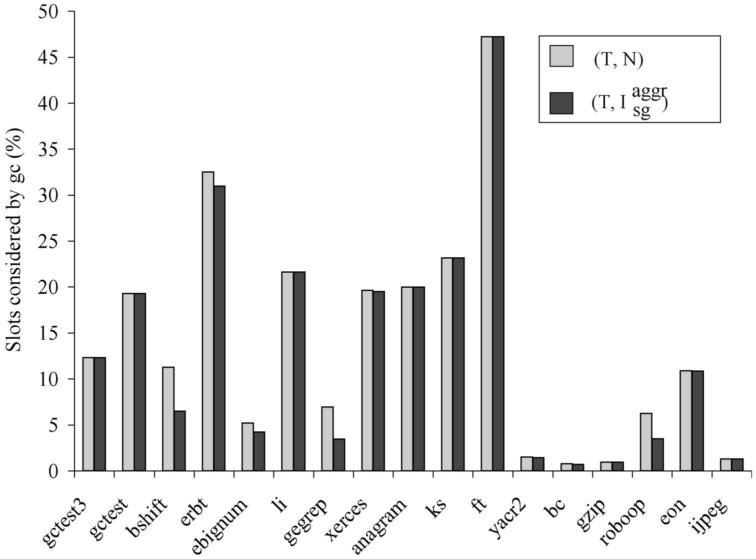

Fig. 12. Percentage of heap slots that the garbage collector must examine.

accurate collectors than with conservative collectors. We see that programs for which liveness accuracy reclaims many more bytes than type accuracy (e.g., *bshift* in Figure 8) are also the ones where there is the most difference between the (T, N) and $\left(T, I_{sg}^{\text{aggr}}\right)$ bars in Figure 12.

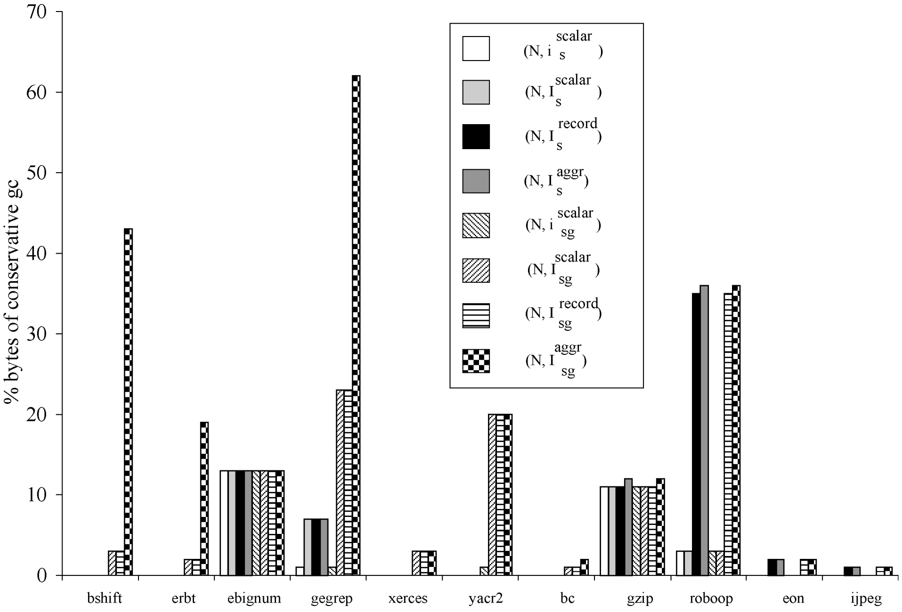

Fig. 13. How much liveness do we need (average)? We omit benchmarks for which there is no benefit from liveness.

6.3 Strength of Liveness Analysis

The prior sections presented results for our most aggressive liveness schemes, $(T, I_{sg}^{\text{aggr}})$ and $(N, I_{sg}^{\text{aggr}})$. Since more accurate liveness information is more difficult to implement and expensive to compute, it is important to determine the point of diminishing return for liveness. In this section, we investigate how powerful the liveness analysis needs to be before it is useful.

Figure 13 gives the impact of the liveness accuracy on the garbage collector or leak detector's ability to identify dead objects. For each benchmark, it has 8 bars. Each bar gives the result for one level of liveness. The first four bars give the benefit of liveness analysis for stack variables only and the next four bars give the benefit of liveness analysis for global and stack variables. Figure 14 is similar to Figure 13 except that it gives the benefit of different levels of liveness accuracy at the point where the number of live bytes in the conservative scheme is at its maximum. Thus, Figure 14 gives a sense of the maximum memory size reduction due to liveness accuracy when compared to conservative collection.

Figures 13 and 14 present results only for liveness levels that offer some benefit (Section 5.1). Also, Figures 13 and 14 present data only for benchmarks where liveness accuracy is useful.

From Figure 13, we see that intraprocedural stack liveness affects only four benchmarks (*ebignum*, *gegrep*, *gzip*, and *roboop*) and the benefits are small. This is consistent with behavior observed by Agesen et al. [1998]. Adding interprocedural analysis to analysis of local variables slightly improves the results for *gegrep* and adding analysis of records and aggregates significantly improves the results for *roboop*. Note that doing just interprocedural analysis of stack

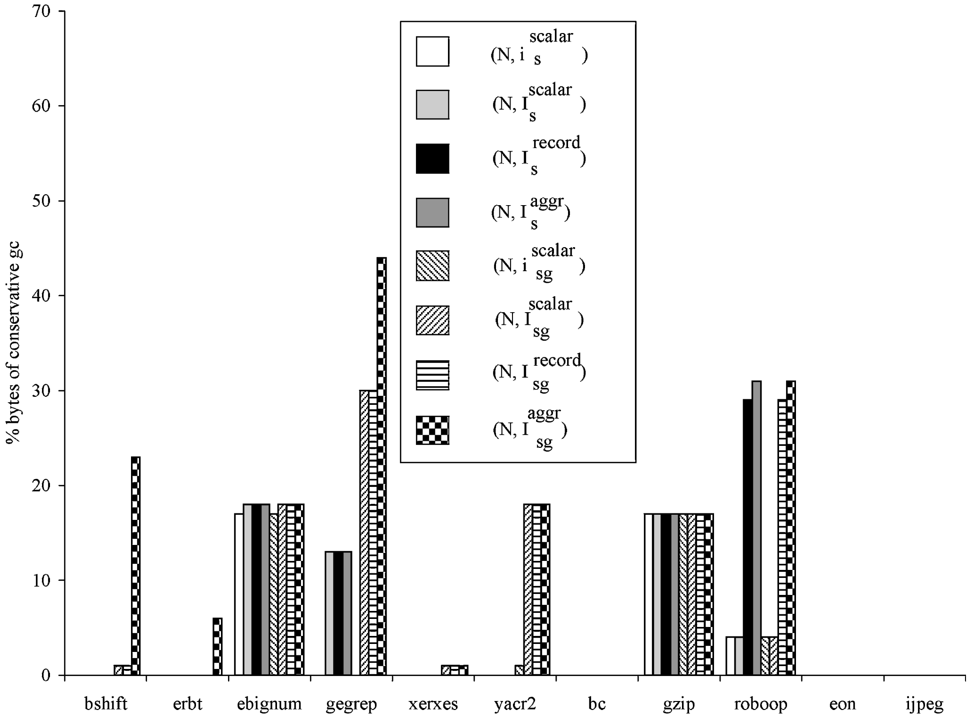

Fig. 14. How much liveness do we need (maximum)? We omit benchmarks for which there is no benefit from liveness.

scalars or intraprocedural analysis of local aggregates[12] does not improve performance for *roboop* but doing both has a synergistic effect.

The majority of the benefit of liveness analysis comes from analyzing global variables (see second set of four bars in Figure 13). 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 and analyzes records. The cumulative impact of 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 intraprocedurally is 0%, but the benefit of adding both is 43%.

Figure 15 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 that the fields of *s* are used consistently with their types. If we analyze procedure f using an interprocedural analysis without records then we would have to conclude that the two fields of *s* may contain live

[12] We do not present results for this configuration since it performs the same as $\left(N, i_s^{\text{scalar}}\right)$.

```
struct{ int *i; int *j;}s;
void f(){
    g();
}
```

Fig. 15. Example of the synergy between analyzing aggregates and doing interprocedural analysis.

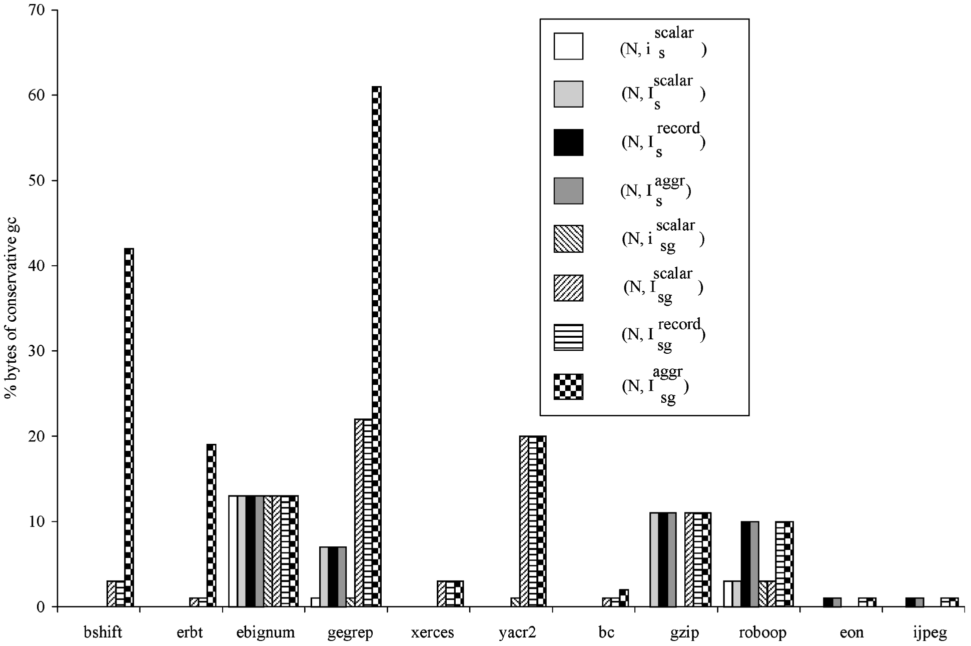

Fig. 16. How much liveness do we need (assuming worst case about library calls)?

pointers at the call to g since the analysis is conservative about record fields (it assumes all record fields are always live). If we analyze procedure f using an intraprocedural liveness analysis that analyzes records, 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 live pointers.

For many programs, $(N, I_{sg}^{\text{record}})$ and $(N, I_{sg}^{\text{aggr}})$ have similar performance (*ebignum*, *yacr2*, *bc*, *gzip*, *ijpeg*, *eon*, and *roboop* plus all the benchmarks that do not benefit from liveness), which is encouraging since $(N, I_{sg}^{\text{record}})$ requires analysis of only records while $(N, I_{sg}^{\text{aggr}})$ requires analysis of records and arrays. Arrays are much harder to analyze and it is unlikely that any reasonable compiler analysis would approach the performance of $(N, I_{sg}^{\text{aggr}})$.

For the above results, we assumed that all liveness analyses, intraprocedural and interprocedural, had information that allowed them to precisely analyze library calls. Figure 16 presents results similar to Figure 13 except that we computed these results while assuming the worst case about standard libraries. Comparing Figures 16 and 13 we see that precise information about library routines makes a significant difference in one program, *roboop*. Information about library routines also helps five other programs to a smaller extent (*bshift*, *erbt*, *gegrep*, *gzip*, and *eon*).

6.4 Type Accuracy and Underlying Architecture

From Section 2, we know that the usefulness of type information for reclaiming objects depends on what pointers look like. Since the appearance of pointers depends on memory layout and on the underlying architecture (e.g., 32-bit or 64-bit), type information may have a different benefit if the program is run using a different memory layout or architecture.

To investigate the effects of memory layout on the usefulness of type accuracy, we ran our programs four times each with a different heap starting address (0x10000000, 0x20000000, 0x40000000, and 0x80000000). We observed small variations in our four runs but even then the benefit of type accuracy for reclaiming objects was small: no more than 3% for any of our programs and 0% for most of the programs. The run with starting address of 0x20000000 showed the most improvement from type accuracy.

To investigate the effects of architecture on the usefulness of type accuracy, we ran several of our benchmarks on a 64-bit Alpha workstation running OSF and a 32-bit SPARC workstation running Solaris. On the SPARC, we found type accuracy to be more important than on the Pentium. Type accuracy improved the effectiveness of garbage collection for several benchmarks, including *gctest*, *bc*, *ft*, *yacr2*, *bshift*, *li*, *gzip*, and *ijpeg*. Some benchmarks showed major improvement: 25% (*gzip*) and 35% (*ijpeg*). We suspect that some of the image data in *ijpeg* is being interpreted as pointers and therefore garbage collection of *ijpeg* benefits greatly from type accuracy. On the Alpha, the improvement due to type accuracy was even less than on the Pentium, with only one benchmark, *gzip*, showing non-trivial benefit from type accuracy.

Our results suggest an idea for strengthening leak detectors: running a program using a leak detector (such as Purify [Hastings and Joyce 1992]) multiple times with different starting memory addresses and on different architectures. The combined information from multiple runs will expose more leaks than a single run.

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. Also, the runs we use are short since our liveness analyses are quite slow. In this section we present numbers for a second run of several benchmark programs. We tried to pick runs that were longer and performed more allocation than our original runs: *bshift* (104 248 bytes), *gegrep* (453 756 bytes), *gzip* (20 240 bytes), *roboop* (1 929 864 bytes), and *yacr2* (76 376 bytes).

Table X gives the stack and global locations that are different between the two runs as a percentage of total stack and global locations when using $\left(N, I_{sg}^{\mathrm{aggr}}\right)$. If

Table X. Number of Stack and Global Locations That are Different as a Percentage of Total Static Stack and Global Locations

	Stack		Global	
Benchmark	Count	% different	Count	% different
bshift	1574	0.2	10469	0
gegrep	19327	0.4	34220	0
gzip	2075	3.4	84158	0.6
roboop	46330	0	10970	0
yacr2	586	3.0	384	0

a stack or global location had a different liveness at any point in the two runs, we counted that location as "different". The total stack and global locations are in the *Count* columns. The results for other levels of accuracy are similar or better.

From Table X, we see that there is little difference between the liveness information for our two runs. To see whether these small differences translate into different behavior, we also measured the retained bytes of $\left(T, I_{sg}^{\text{aggr}}\right)$ and $\left(N, I_{sg}^{\text{aggr}}\right)$. 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 garbage was of course different between the two runs. Thus, it is likely that our run-time methodology is computing a tight approximation.

6.6 Summary of Results

To summarize, our results show that as far as reclaiming objects in our benchmark runs is concerned, type accuracy or weak liveness accuracy schemes, such as ones implemented in current systems, yield little or no benefit. Type accuracy does, however, reduce the work of the reachability traversal by allowing it to ignore most memory slots. The benefits of type accuracy depend greatly on the memory layout and the underlying hardware architecture: we found that the benefit of type information in reclaiming objects is much greater on a SPARC 32-bit workstation than on the Pentium-based workstations we used for the majority of this study. Aggressive liveness analyses, particularly ones that analyze globals records and are interprocedural, promise significant improvements.

Figure 17(b) shows the experimental relationship between the accuracy levels. Figure 17(b) is a subset of Figure 6 containing only the solid nodes, but with a different interpretation of vertical position (the horizontal position has no significance). For each scheme S in Figure 17(b), the vertical position corresponds to the metric $\text{avg}\frac{(N,N)-S}{(N,N)}\%$, which is explained in Section 5.2. The horizontal lines in Figure 17(b) $\left(\text{e.g., between } \left(T, I_{sg}^{\text{aggr}}\right) \text{ and } \left(N, I_{sg}^{\text{aggr}}\right)\right)$ connect accuracy schemes that differ in strength only theoretically but insignificantly in our experiments.

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 [Boehm et al. 2002]. The BDW

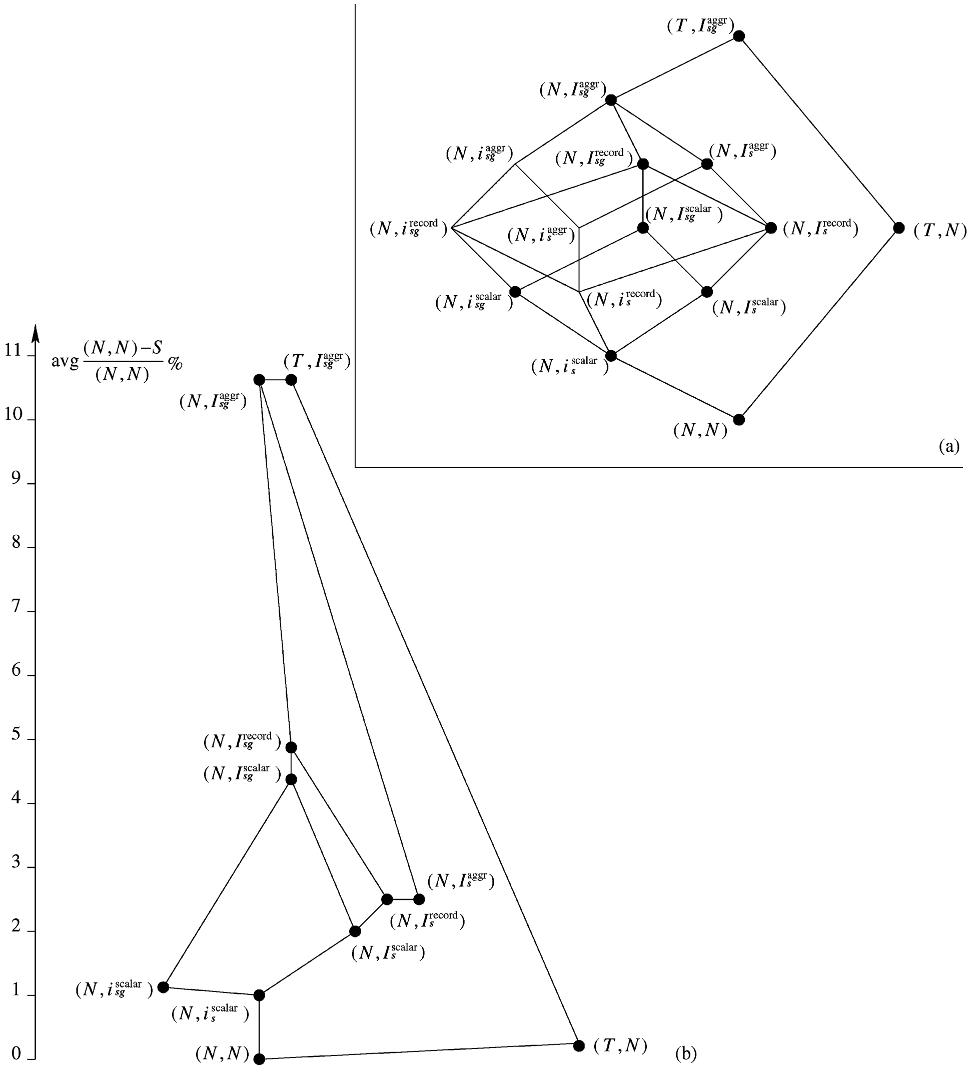

Fig. 17. Theoretical (a) and experimental (b) strength of accuracy schemes.

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 [Dion and Monier 2002]. 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 an in-use object as garbage 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 an object that is still in use 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 be able to tell whether the leak is due to a bug in the garbage collector or whether it is due to an unfortunate pointer in their own code. All bugs we found in the BDW collector were of kind (ii).

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 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 that 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. 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.

Röjemo and Runciman [1996] describe heap profiling, which they use to tune their applications. Röjemo and Runciman's approach is to provide heap profile information to programmers who can use it to tune the memory usage of their applications. Röjemo and Runciman introduce the notion of "drag", which is the time between the last real use of an object and the time at which it is deallocated. Their notion of "drag" is stronger than our strongest liveness analysis since it is derived from exact information on when a heap object is last accessed. Our basic goals are similar: we are also trying to reduce the memory footprint of programs and are using profiling techniques to accomplish our goals. However, for Röjemo and Runciman, the profiling tools provide information to programmers whereas, for us, they provide information to designers of memory managers.

In work concurrent to ours, Shaham et al. [2000] 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 demonstrate if an accurate collector would be more effective in reclaiming objects. In their follow-up work, Shaham et al. [2001]

investigate manual optimizations (such as dead-code elimination and insertion of nil assignments) to improve the performance of their garbage collector; these optimizations were guided by their limit study. Shaham et al. [2001] also discuss which compiler analyses would be needed to automate their optimizations. Thus, Shaham et al. [2001] and our work have similar goals. Our work differs from Shaham et al.'s [2001] work in that we determine the needed compiler analyses automatically rather than manually.

Bartlett [1988], Zorn [1993], Smith and Morrisett [1998], and Agesen et al. [1998] compare different garbage collection alternatives with respect to memory consumption. Bartlett [1988] describes versions of his mostly copying garbage collector that differ in stack accuracy. Zorn [1993] compares the Boehm–Demers–Weiser collector to a number of explicit memory management implementations. Smith and Morrisett [1998] 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. [1998] 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.

Several papers (Attanasio et al. [2001], Fitzgerald and Tarditi [2000], Hicks et al. [1997], Smith and Morrisett [1998], and Zorn [1993]) compare different memory management schemes with respect to their efficiency. Zorn [1991] looks at the cache performance of different garbage collectors. We do not look at runtime efficiency but instead concentrate on the effectiveness of garbage collectors in reclaiming objects.

Boehm and Shao [1993] describe a technique for improving 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.

Henderson [2002] describes how to compile type-safe languages to C without giving up type accuracy. The resulting C program, which can be compiled to machine code using a conventional compiler, can safely use a copying garbage collector.

Diwan et al. [1992], Agesen et al. [1998], and Stichnoth et al. [1999] consider how to perform accurate garbage collection in compiled type-safe languages. Diwan et al. [1992] describe how the compiler and run-time system of Modula-3 can support accurate garbage collection. Agesen et al. [1998] and Stichnoth et al. [1999] extend Diwan et al.'s work by incorporating liveness 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 [1992], Dion and Monier [2002], and Geodesic Systems [2002] describe leak detectors based on the Boehm–Demers–Weiser collector [Boehm and Weiser 1988]. The Boehm–Demers–Weiser collector can also be used as a leak detector [Boehm et al. 2002]. Our scheme uses more accurate information than these leak detectors, and is thus capable of finding more leaks in programs.

Region-based memory management [Tofte 1998] uses a compile-time liveness analysis of objects to determine where to automatically insert deallocations in the code. In this paper we focus only on liveness analysis of local and global variables and not heap-allocated objects.

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. Our novel methodology, which uses a trace-based analysis, enables us to 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 for our benchmark runs: we need to use an aggressive liveness scheme that incorporates interprocedural analysis of global variables before we see a significant benefit. Also, in our benchmark runs, we found that, while type accuracy can significantly reduce the work of the reachability traversal, it has a negligible impact on a garbage collector's ability to reclaim objects.

ACKNOWLEDGMENTS

We thank Michael Hind, Urs Hölzle, Eliot Moss, and our anonymous referees for ISMM 2000, ECOOP 2001, and TOPLAS for comments on this research, and John DeTreville, Tony Hosking, Dirk Grunwald, and Alex Wolf for fruitful discussions about the ideas in the article.

REFERENCES

AGESEN, O., DETLEFS, D., AND MOSS, J. E. B. 1998. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *Programming Language Design and Implementation* (*PLDI*). ACM, New York, pp. 269–279.

ALPERN, B., ATTANASIO, C. R., BARTON, J. J., BURKE, M. G., CHENG, P., CHOI, J.-D., COCCHI, A., FINK, S. J., GROVE, D., HIND, M., HUMMEL, S. F., LIEBER, D., LITVINOV, V., MERGEN, M. F., NGO, T., RUSSELL, J. R., SARKAR, V., SERRANO, M. J., SHEPHERD, J. C., SMITH, S. E., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. 2000. The Jalapeño virtual machine. *IBM Syst. J.*, *39*, 1 (Feb.), 211–238.

APPEL, A. W. 1990. A runtime system. *LISP Symb. Computat.*, *3*, 4 (Nov.), 343–380.

ATTANASIO, C. R., BACON, D. F., COCCHI, A., AND SMITH, S. 2001. A comparative evaluation of parallel garbage collector implementations. In *Workshop on Languages and Compilers for Parallel Computing* (*LCPC*), (Aug.).

BARTLETT, J. F. 1988. Compacting garbage collection with ambiguous roots. Tech. Rep. 88/2. DEC Western Research Laboratory, Palo Alto, Calif. (Also in *LISP Pointers 1*, 6 (Apr.-June), 2–12.

BARTLETT, J. F. 1989. Mostly-copying garbage collection picks up generations and C++. Tech. Rep. DEC Western Research Laboratory, Palo Alto, Calif.

BOEHM, H.-J., DEMERS, A. J., AND SHENKER, S. 1991. Mostly parallel garbage collection. In *Programming Language Design and Implementation* (*PLDI*) (Nov.). ACM, New York, pp. 157–164.

BOEHM, H.-J., DEMERS, A. J., AND WEISER, M. 2002. A garbage collector for C and C++. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

BOEHM, H.-J. AND SHAO, Z. 1993. Inferring type maps during garbage collection. In *OOPSLA '93 Workshop on Memory Management and Garbage Collection* (Sept.). ACM, New York.

BOEHM, H.-J. AND WEISER, M. 1988. Garbage collection in an uncooperative environment. *Softw.—Pract. Exp.* (*SPE*), *18*, 9 (Sept.), 807–820.

COLNET, D. COUCAUD, P., AND ZENDRA, O. 1998. Compiler support to customize the mark and sweep algorithm. In *International Symposium on Memory Management* (*ISMM*) (Oct.), pp. 154–165.

DION, J. AND MONIER, L. 2002. Third degree. `http://research.compaq.com/wrl/projects/om/third.html`.

DIWAN, A., MOSS, J. E. B., AND HUDSON, R. L. 1992. Compiler support for garbage collection in a statically typed language. In *Programming Language Design and Implementation* (*PLDI*) (July). ACM, New York, pp. 273–282.

EDISON DESIGN GROUP. 2002. `http://www.edg.com`.

FITZGERALD, R. AND TARDITI, D. 2000. The case for profile-directed selection of garbage collectors. In *International Symposium on Memory Management* (*ISMM*) (Oct.), pp. 111–120.

GEODESIC SYSTEMS. 2002. Great Circle—Real-time error detection and code diagnosis for developers. `http://www.geodesic.com/solutions/products_gc_overview.html`.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison-Wesley, Reading, Mass.

HASTINGS, R. AND JOYCE, B. 1992. Fast detection of memory leaks and access errors. In *Proceedings of the Winter '92 USENIX Conference*, pp. 125–136.

HENDERSON, F. 2002. Accurate garbage collection in an uncooperative environment. In *International Symposium on Memory Management* (*ISMM*) (June).

HICKS, M. W., MOORE, J. T., AND NETTLES, S. M. 1997. The measured cost of copying garbage collection mechanisms. In *International Conference on Functional Programming* (*ICFP*) (June), pp. 292–305.

HUDSON, R. L., MOSS, J. E. B., DIWAN, A., AND WEIGHT, C. F. 1991. A language-independent garbage collector toolkit. Tech. Rep. 91-47. Univ. Massachusetts at Amherst, Amherst, Mass.

JONES, R. AND LINS, R. 1997. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, New York.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The definition of Standard ML*. MIT Press, Cambridge, Mass.

NELSON, G. 1991. (Ed.) *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, N.J.

RÖJEMO, N. AND RUNCIMAN, C. 1996. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In *International Conference on Functional Programming* (*ICFP*) (Philadelphia, Pa. May) pp. 34–41. (Also available as *ACM SIGPLAN Notices 31*, 6).

SHAHAM, R., KOLODNER, E. K., AND SAGIV, M. 2000. On the effectiveness of GC in Java. In *International Symposium on Memory Management* (*ISMM*) (Oct.), pp. 12–17.

SHAHAM, R., KOLODNER, E. K., AND SAGIV, M. 2001. Heap profiling for space-efficient Java. In *Programming Language Design and Implementation* (*PLDI*) (June). ACM, New York, pp. 104–113.

SMITH, F. AND MORRISETT, G. 1998. Comparing mostly-copying and mark-sweep conservative collection. In *International Symposium on Memory Management* (*ISMM*) (Oct.), pp. 68–78.

STANFORD UNIVERSITY SUIF RESEARCH GROUP. 2002. SUIF compiler system version 1.x. `http://suif.stanford.edu/suif/suif1/index.html`.

STICHNOTH, J. M., LUEH, G.-Y., AND CIERNIAK, M. 1999. Support for garbage collection at every instruction in a Java compiler. In *Programming Language Design and Implementation* (*PLDI*) (May). ACM, New York, pp. 118–127.

Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P. 1996. TIL: A type-directed optimizing compiler for ML. In *Programming Language Design and Implementation* (*PLDI*) (May). ACM, New York, pp. 181–192.

Tofte, M. 1998. A brief introduction to regions. In *International Symposium on Memory Management* (*ISMM*) (Oct.), pp. 186–195.

Ungar, D. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments* (*SDE*). pp. 157–167.

Wilson, P. 1992. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management* (Sept.), pp. 1–42.

Wilson, R. P., French, R. S., Wilson, C. S., Amarasinghe, S. P., Anderson, J.-A. M., Tjiang, S. W. K., Liao, S.-W., Tseng, C.-W., Hall, M. W., Lam, M. S., and Hennessy, J. L. 1994. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Not. 29*, 12 (Dec.), 31–37.

Zorn, B. 1993. The measured cost of conservative garbage collection. *Softw.—Pract. Exp. 23*, 7 (July), 733–756.

Zorn, B. G., 1991. The effect of garbage collection on cache performance. Tech. Rep. CU-CS-528-91. Univ. Colorado at Boulder, Boulder, Col., May.

Received September 2001; accepted May 2002