On the Type Accuracy of Garbage Collection

Martin Hirzel
University of Colorado
Boulder, CO 80309
hirzel@cs.colorado.edu

Amer Diwan
University of Colorado
Boulder, CO 80309
diwan@cs.colorado.edu

ABSTRACT

We describe a novel approach to obtaining type-accurate information for garbage collection in a hardware and language independent way. Our approach uses a run-time analysis to propagate pointer/non-pointer information from significant type events (such as allocation, which always returns a pointer). We use this technique to perform a detailed comparison of garbage collectors with different levels of accuracy and explicit deallocation on a range of C programs. We take advantage of the portability of our approach to conduct our experiments on three hardware platforms, Alpha/Digital UNIX 4.0D, Pentium/Linux 2.2, and SPARC/Solaris 2. We find that the choice of hardware platform (which includes the architecture, operating system, and libraries) greatly affects whether or not type accuracy enhances a garbage collector's ability to reclaim objects.

1. INTRODUCTION

Garbage collection (GC), or automatic storage reclamation, has well-known software engineering benefits [16]. 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 garbage collection to free "dead" objects [6, 3] or identify memory leaks in programs that use explicit deallocation [9, 7, 5, 1]. In order for a garbage collector to be effective for these two uses, it must satisfy the following requirements. First, the garbage collector must be competitive with a careful programmer at collecting garbage. If the garbage collector fails to identify dead objects then it will either cause memory leaks or fail to identify memory leaks, depending on how garbage collection is being used. Second, the garbage collector must be competitive in run-time performance with explicit deallocation. The second requirement is less important if the garbage collector is used as a leak detector. This paper evaluates garbage collectors with different levels of accuracy and explicit deallocation with respect to the first requirement.

One factor that determines a garbage collector's effectiveness in finding dead objects is *type accuracy*. Type accuracy determines whether or not a garbage collector can distinguish pointers from non-pointers in memory. The two extremes of type accuracy are *fully type accurate* and *conservative*. A fully type accurate garbage collector can correctly distinguish pointers from non-pointers in all regions of memory (globals, locals, and heap). A conservative garbage collector cannot *reliably* distinguish pointers in any region of memory. Between these two, there are partially-accurate garbage collectors that can distinguish pointers from non-pointers reliably in some regions of memory but not in others. If a garbage collector is not accurate for a particular region of memory it must use heuristics to identify pointers in that region. In a nutshell, the collector must treat all values in that region that look like pointers as pointers. Since different hardware platforms (which includes architecture, operating system, and libraries) use different memory layouts and alignments, a value that looks like a pointer on one machine may not look like a pointer on another machine.

This paper examines the effect of type accuracy on garbage collection effectiveness for a range of C benchmark programs. We consider all combinations of type accuracy (accuracy in stack, globals, or heap) and compare them to the original explicit memory management in the C benchmark programs. We use a novel run-time analysis that tracks the flow of pointers through memory locations to compute accuracy information for C programs. Finally, we present our measurements for three hardware platforms that use widely different memory layouts.

Our results demonstrate that the benefit of type accuracy in reclaiming objects depends greatly on the hardware platform. On the Alpha/UNIX 4.0D, a 64-bit machine, all garbage collectors (including conservative collectors) are competitive with explicit deallocation in reclaiming objects. On 32-bit machines, such as the Pentium/Linux 2.2 and the SPARC/Solaris 2, type accuracy significantly affects the effectiveness of a garbage collector: accurate garbage collectors continue to perform well on 32-bit machines but the less precise collectors perform relatively poorly. We show that the relative importance of accuracy in different regions of memory also depends on the hardware platform. On the Alpha and the Pentium accuracy in the stack and global areas is the most important. On the SPARC accuracy in the heap is just as important as accuracy in other areas of memory.

The rest of this paper is organized as follows. Section 2 gives an introduction to type accuracy in garbage collectors and discusses the memory layout of three hardware platforms. Section

3 describes our benchmark programs, evaluation methods, and our novel approach to type accuracy. Section 4 presents the experimental results. Sections 5 and 6 present related work and possible directions for future work. Section 7 concludes the paper.

2. BACKGROUND

Broadly speaking there are two kinds of garbage collectors: *type accurate* and *conservative*. A *type accurate* garbage collector knows exactly which memory locations and registers contain pointers. Type-accurate collectors typically require compiler support in the form of tables or code that identifies pointers [8, 14]. In contrast, a *conservative* garbage collector cannot reliably identify pointers: it assumes that every value in memory that could be interpreted as a pointer is indeed a pointer [6]. Conservative garbage collectors require little or no compiler or language support; indeed this is one of the main selling points of conservative garbage collection.

Type accurate and conservative collectors are at two extremes of type accuracy; one can imagine collectors that fall in between the two. For instance, Bartlett's collector [3, 4] is type accurate for the heap but inaccurate for other areas of memory. Other collectors with intermediate accuracy, such as the ones that are type accurate just for the globals, are possible but, to our knowledge, have not been explored in literature.

The main advantage of conservative garbage collection over type-accurate collection is that it can be used in almost any language environment with little effort, since it requires little or no compiler or language cooperation. However, prior literature (*e.g.*, Chapter 9 of Jones' book [11]) points out many deficiencies of conservative collection compared to accurate collection. Two main deficiencies are:

- Unlike type-accurate collection, a conservative collector cannot compact objects. Compaction of live objects improves memory locality, enables fast allocation, and eliminates fragmentation. Thus, conservative collection can register worse performance than a compacting type-accurate collector.
- A conservative collector may misidentify a pointer, thus preventing garbage objects from being deallocated.

Prior work has partly investigated the first deficiency of conservative garbage collection [13]. To our knowledge, no one has done a quantitative and direct study of the second deficiency of conservative garbage collection.

Table 1 describes what pointers look like in three different hardware platforms:[1] *SPARC 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 [5] during a run of one of our benchmark programs (*bc*).[2] The table also shows what that address maps 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. Note that in the "int" and "float" rows for Alpha there are two values: this is 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 as a pointer. When pointers are interpreted as integers, we see that on the Alpha two adjacent and aligned 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.[3] 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.

3. METHODOLOGY

We conducted our study by measuring the behavior of eleven C programs with different memory managers ranging from explicit deallocation to type-accurate garbage collection. We ran our experiments on three hardware platforms—Alpha, Pentium, and SPARC—to investigate the platform-dependent effects outlined in Section 2. We start this section by describing our benchmark programs and giving basic statistics about the programs (Section 3.1). We then describe the different memory management schemes that we explored (Section 3.2).

3.1 Benchmarks

We used two 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.

Table 2 describes our benchmark programs. *Lines* gives the number of lines in the source of the program (including comments and blank lines). Two of our benchmarks, *gctest* and *gctest3*, are designed to test garbage collectors [3, 4]. 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 based on our policy (Section 3.2.3). *bshift* is an Eiffel program that was translated into C using the Eiffel-to-C compiler distributed as part of GNU SmallEiffel. *li* and *bshift* include custom garbage collectors. We replaced calls to these custom collectors with calls to our collector. The remaining programs use standard C allocation and deallocation to manage memory.

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

[2]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.

[3]Except, of course, if integers are used to implement bit vectors.

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

Table 1: Pointers in three hardware platforms

Name	Source	Lines	Main data structures	Dealloc.	Workload	Kind of program
gctest3	Bartlett	85	lists and arrays	automatic	loop to 20,000	synthetic test
gctest	Bartlett	196	lists and trees	automatic	only repeat 5 in listtest2	synthetic test
anagram	Austin	647	lists and bitfields	explicit	`words < input.in`	string processing
ks	Austin	782	D-arrays and lists	explicit	`KL-2.in`	graph algorithm
ft	Austin	2 156	graphs	explicit	`1000 2000`	graph algorithm
yacr2	Austin	3 979	arrays of structures	explicit	`input2.in`	channel router
bshift	Hirzel	4 398	dlists	automatic	scales 2 through 12	object-oriented
bc	Austin	7 308	abstract syntax trees	explicit	find primes smaller 500	calculator/interpreter
li	Spec95	7 597	cons cells	automatic	`boyer.lsp`	functional/interpreter
gzip	GNU	8 163	Huffman trees	explicit	`-d texinfo.tex.gz`	compression
ijpeg	Spec95	31 211	various image repn.	explicit	`testinput.ppm`	image compression

Table 2: Benchmarks

Table 3 shows the number of bytes each benchmark allocates throughout its execution on each of three hardware platforms.

Name	Alpha	Pentium, SPARC
gctest3	3 600 008	2 200 004
gctest	1 702 376	1 123 180
anagram	265 984	259 512
ks	15 840	7 920
ft	298 056	166 832
yacr2	267 512	148 680
bshift	1 189 976	793 132
bc	10 854 848	12 382 400
li	7 669 920	4 792 160
gzip	28 376	14 180
ijpeg	11 326 128	9 030 872

Table 3: Total allocation in bytes

3.2 Memory management schemes

We now describe the memory management schemes we explored: explicit memory management and garbage collection with various levels of accuracy. We conclude this section by discussing how we compared the different schemes.

3.2.1 Explicit memory management

We conducted our experiments with explicit memory management (hereafter referred to as *free*) only for those programs that used explicit memory management to start with. In these programs we inserted wrappers around allocation and deallocation routines to track various memory allocation and deallocation statistics.

3.2.2 Garbage collection

We conducted our experiments with garbage collection by replacing calls to allocation routines with calls to the garbage collector allocator and by eliminating calls to deallocation routines. The garbage collector is the Boehm-Demers-Weiser collector [5] modified to accept accurate type information about one or more regions of memory.

Since C programs may violate type declarations, we cannot use traditional mechanism for providing accurate information to the BDW collector. For instance, imagine a program that uses an appropriately sized integer to hold a pointer. Clearly this “integer” should be considered by the garbage collector as a pointer even though its programming language type is not a pointer. We therefore use a new approach to gathering accurate type information about C programs. The key to our approach is to determine type information at run time when we have more precise knowledge about what locations contain pointers and what locations contain non-pointers. To accomplish this, we run the program twice, one time to collect type information and the second time to use the information for accurate garbage collection. Of course the two runs must use the same set of inputs in order for the type information from the first run to hold for the second run. We now describe the mechanics of our two runs in more detail and conclude this section by discussing the strengths and weaknesses of our approach.

Figure 1 illustrates our approach to type accuracy for C programs. The first step (*instrumentor*) adds calls to the C program to mark every event that may update type (or rather pointer/non-pointer) information. We also instrument the code of any library routines that the C program calls which may manipulate pointers in the user data (such as *memcpy*). Table 4 shows examples of the calls we add to the C code. Basically, we insert calls that identify the target and the sources of every assignment. The call sequence for each assignment begins by identifying the target of the assignment and then has a call for each value used to compute the value being assigned. If a value is definitely a pointer (e.g., *&y* or *malloc(4)*) then we call *note_ptr_assignment_source* otherwise we call *note_assignment_source*. We treat call parameters

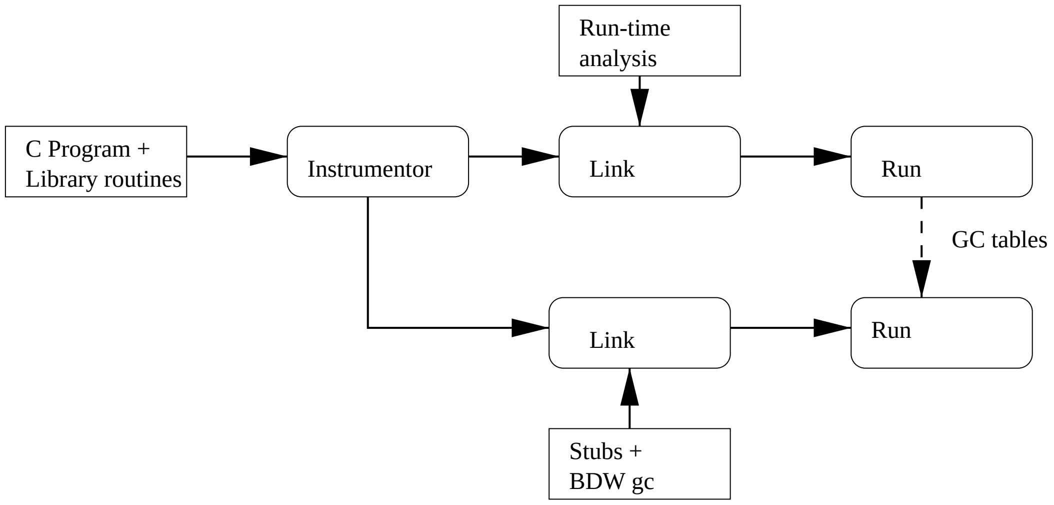

Figure 1: Framework for collecting type-accurate information

C Expression	Calls added
x = y	note_assignment_target(&x) note_assignment_source(&y)
*x = y	note_assignment_target(x) note_assignment_source(&y)
x = &y	note_assignment_target(&x) note_ptr_assignment_source()
x = y op z	note_assignment_target(&x) note_assignment_source(&y) note_assignment_source(&z)
x = malloc(4)	note_assignment_target(&x) note_ptr_assignment_source()
int global	note_global(&global)
int local	note_stack_alloc(&local)

Table 4: Examples of calls added to C code

C Expression	Action
x = y	is_ptr(x) = is_ptr(x) or is_ptr(y)
*x = y	is_ptr(*x) = is_ptr(*x) or is_ptr(y)
x = &y	is_ptr(x) = true
x = y op z	is_ptr(x) = is_ptr(x) or is_ptr(y) or is_ptr(z)
x = malloc(4)	is_ptr(x) = true;

Table 5: Outline of run-time analysis

and returns similarly to assignments. In addition to this instrumentation, we replace calls to each allocation routine *alloc* with calls to *my_alloc*, which takes the same parameters as *alloc*. Our instrumentation pass uses the SUIF compiler infrastructure [15] and can be used for any language as long as there is a translator from the language into SUIF or ANSI C.

We get our data-collection run by linking the instrumented program with a *run-time analysis library*. This library provides implementation for all routines called by the instrumentation. These implementations perform a run-time type analysis interleaved with the original program execution. Table 5 outlines at a high level the actions that our run-time analysis takes. From this table we see that once our analysis determines that a variable contains a pointer, it assumes that the variable always contains a pointer even after a non-pointer value is assigned to it. This can result in imprecision in our analysis if a program uses a variable to hold pointers and non-pointers at different times during the execution.

At the end of this run, the program outputs tables that describe which memory locations contain pointers. The second run uses these tables to provide accurate information to the garbage collector. Since memory addresses of objects may be different in the second run, the first run assigns unique identifiers to each heap-allocated object and global variable and uses these identifiers to refer to objects. Each entry in the table is of one of the following forms:

- (global_id, offset): the global variable identified by global_id contains a pointer at *offset*.
- (heap_id, offset): the heap allocated object identified by heap_id contains a pointer at *offset*.
- (proc_name, offset): activation records for the procedure identified by proc_name contain a pointer at *offset*.

Note that 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.

To generate the executable for the second run we link the instrumented program with a modified BDW collector. We modified the BDW collector to use type information *when available*. If accurate type information is not available for a region of memory then BDW uses its own pointer identification mechanisms to identify potential pointers in that region. Even if type information is available for a region, BDW still checks that information with its own pointer identification mechanism since the type information may not be precise (for example, if a program uses a variable to hold a pointer at some points and a non-pointer at other points). The second run uses the instrumented rather than the original program to ensure that both runs have the same stack layout. However, unlike the first run which uses a "run-time analysis" library, the second run uses a "stubs" library that implements all analysis routines with empty procedures.

During the second run we collect detailed statistics about live objects after each explicit *free* and after each garbage collection. For this statistics collection run, we can choose to either use or ignore each of local, heap, and global accuracy individually, and we can choose to allow or ignore explicit deallocation. Finally, we force garbage collections to occur at exactly the same points in the program execution in all runs (see Section 3.2.3).

The main advantage of our approach is that it is language independent: we can use it to evaluate the effectiveness of accuracy for any program that can be compiled to the SUIF representation. However, since it requires two runs (which are both quite slow), it cannot provide real accurate garbage collection—it is just a method for measuring one aspect of accuracy: object retention. In particular, we cannot use this approach yet to measure other benefits of accuracy (such as compaction or fast allocation). We conclude this section by discussing the limitations of our approach or current implementation:

- When a location in an object contains a pointer at any time during the program run, we identify it as a pointer at all times during program execution. If the programmer happens to store a non-pointer that looks like a pointer in the same location, the garbage collector may wrongly retain memory.
- BDW uses not just the activation records belonging to user routines but also some of its own activation records as roots for garbage collection. Since we do not have precise type information for BDW's own activation records, we have to be conservative for these activation records. We think that we can avoid using these activation records as roots but have not investigated this so far.
- If at least one of the operands of an arithmetic expression is a pointer, we identify the result as a pointer. We expect that this may cause incorrect retention of objects if the result of an expression looks like a pointer but will never actually be used like a pointer (for example if the arithmetic expression is computing the hash value of a pointer). We are in the process of reducing this imprecision in our implementation.
- While our implementation currently handles most of C (including *setjmp/longjmp*) it does not yet support programs that use signal handlers, call *exit* or *setjmp/longjmp* through function pointers, or use *alloca*. We believe it is relatively straightforward to extend our implementation to support these features.

3.2.3 *Comparison methodology*

To facilitate comparison across garbage collectors with different levels of accuracy we forced garbage collection to be triggered every n bytes of allocation, where n is the same in all configurations. We ran each benchmark twice with each level of garbage collection accuracy. The first time we picked n to trigger approximately 5 garbage collections during the program run and the second time we picked n to trigger approximately 50 garbage collections during the program run. With 5 garbage collection runs the number of live bytes between collection grows to be much larger (by an order of magnitude) than with 50 garbage collections. Thus there is a greater possibility that a non-pointer value in memory will look like a pointer for the 5 garbage collection runs.

We compare explicit deallocation to garbage collection primarily at points immediately following a garbage collection.

3.3 Abbreviations

We now present some abbreviations that we use in the remainder of the paper:

- *gc*: Unmodified BDW collector;
- gc_{shg}: BDW collector using accuracy information (where available) for the stack, heap, and globals;
- *free*: Original program using explicit deallocation.

While the above abbreviations identify different memory management schemes, we will sometimes use them to mean the number of bytes occupied by objects when using the corresponding memory management scheme.

4. RESULTS

This section starts by presenting results that compare garbage collection with explicit deallocation (Section 4.1). It then presents results evaluating the value of different levels of accuracy in garbage collection and leak detectors (Section 4.2). Then it discusses how our approach allows us to build more effective leak detectors than existing ones (Section 4.3). We then try to quantitatively explain why we observe different results for different hardware platforms (Section 4.4). Finally we summarize the results (Section 4.5).

4.1 Garbage collection versus explicit deallocation

Tables 6 and 7 compare accurate (gc_{shg}) and conservative (*gc*) garbage collection to explicit deallocation for the Alpha, Pentium, and SPARC hardware platforms. These tables contain data for all benchmarks that use explicit deallocation. There are two rows for each benchmark, one where we invoked the collector 5 times and one when we invoked the collector 50 times. The *Average* $\frac{gc_{shg}-free}{free}$ column presents an average of the excess bytes that gc_{shg} retains over *free* as a fraction of the bytes retained by *free*. We compute this ratio immediately after each garbage collection and take their arithmetic mean to arrive at the numbers in the table. For example, a value of 0.01 in this column means that if a program used gc_{shg} it would retain, on average, 1% more bytes than explicit deallocation. A negative number in this column means that gc_{shg}, on average, had fewer live bytes than explicit deallocation; in other words, there is a possible memory leak in explicit deallocation (we discuss memory leaks in Section 4.3). The *frac. gc* columns give the fraction of garbage collections after which gc_{shg} had a different number of live bytes than explicit deallocation. The $\frac{gc-free}{free}$ and its accompanying *frac. gc* columns in Table 7 present the same data, but this time comparing *gc* (i.e., BDW garbage collector) to explicit deallocation. In this and subsequent tables, we represent fractions that are smaller than 0.01 but not zero as 0.00 and fractions that are exactly zero as 0.

From these tables we see that for most benchmarks, the number of live bytes using gc_{shg} is close to the number of live bytes for *free* at points immediately following a garbage collection. The two programs where gc_{shg} performs much worse than *free* are *gzip* and *ijpeg*. However, from Table 3 we see that *gzip* does little allocation and thus the difference between gc_{shg} and *free* is not significant. *ijpeg* on the other hand does a significant amount of allocation. Figure 2 compares gc_{shg}, *gc*, and *free* for *ijpeg-5* on a SPARC. A point *(x,y)* on the graph means that at time x (measured in bytes allocated since the start of program), live objects occupy y bytes. From this figure we see that even though *ijpeg* allocates heavily, the number of bytes that are live at any given point is very small. Thus even though gc_{shg} appears to do significantly worse than *free* on *ijpeg* according to Table 6, in absolute terms, the number of excess bytes retained by gc_{shg} is rather small.

From Table 6 we see that the performance of gc_{shg} also varies with hardware platform. This is because gc_{shg} is not fully accurate for C programs (Section 3).

The behavior of conservative garbage collection, *gc*, varies much more across hardware platforms than accurate collection. On the Alpha, *gc* performs almost as well as gc_{shg}. On the Pentium, *gc* does worse (i.e., retains more dead objects) than on the Alpha. On the SPARC, *gc* performs even worse. These results suggest that the choice of hardware platform plays a role in determining which memory management scheme will perform best.

Our results suggest that at least on the Alpha platform, most garbage collectors, including *gc*, will reclaim objects almost as effectively as explicit deallocation. However, this is only part of the picture since it says that garbage collection will have almost the same number of live bytes as explicit deallocation at points *immediately following a garbage collection*. Obviously, at other points, there may be many more live objects with garbage collection. For example Figure 2 shows that immediately following a garbage collection, the number of live bytes is relatively similar for all three memory management strategies. However, at other points, *ijpeg* with garbage collection may have an order of magnitude more live bytes than explicit deallocation. These results make two important points: (i) just because a garbage collected program uses much more memory than a non-garbage collected program, it does not mean that the garbage collector is leaking memory, and (ii) even if a garbage collector is as effective at reclaiming objects as explicit deallocation, it may still have a much larger memory footprint and thus worse memory system behavior. We can decrease the peaks of our garbage collection curves by collecting garbage more frequently, as illustrated by Figure 3 (note that Figure 3 uses a different scale than Figure 2).

Finally Tables 6 and 7 show that for more frequent collections (50 times instead of 5 times) on average the difference between garbage collection and explicit deallocation is greater. This may be caused by dead pointers on the stack; waiting longer between garbage collection allows more activation records to pop off the stack and thus the garbage collection needs to consider fewer dead pointers. Based on this, we expect that adding liveness information to gc_{shg} and *gc* will improve their behavior for more frequent collections [2].

4.2 Effect of garbage collector accuracy

In which region of memory is accuracy most important for each of our three hardware platforms? Table 8 presents data that compares different levels of accuracy to conservative garbage collection. Unlike Tables 6 and 7 which presented data only for programs that use explicit deallocation, Table 8, presents data for our entire benchmark suite. The *Average* $\frac{gc-gc_{shg}}{gc}$ column presents the average of the excess bytes that *gc* retains over gc_{shg} as a fraction of the bytes retained by *gc*. We compute this ratio immediately after each garbage collection and take their arithmetic mean to arrive at the numbers in the table. The characters in parentheses besides the numbers indicate which accuracy contributed to the improvement of gc_{shg} over *gc*. The *frac. gc* column gives the fraction of garbage collections after which gc_{shg} had a different number of live bytes than *gc*.

From these tables we see that accurate garbage collection is slightly better than conservative collection on the Alpha. *yacr2*, *bshift*, and *ijpeg* are the only three benchmarks where gc_{shg} performs significantly better than conservative garbage collection (*gc*). Moreover, for two of the benchmarks, *bshift* and *ijpeg*, gc_{shg} is better than *gc* on only for a relatively small fraction of the time (as evident from the *frac. gc* column).

On the Pentium and especially the SPARC architecture accurate garbage collection does better than conservative collection in the majority of the benchmarks. The most significant improvements from accurate collection come on *yacr2*, *bshift*, *gzip*, and *ijpeg*, but many other benchmarks see smaller improvements due to accuracy. Figures 2 and 3 graphically display the superiority of gc_{shg} over *gc* on *ijpeg*. Figure 4 magnifies the initial segment of Figure 3 to make it clearer.

On inspecting the parenthesized characters next to the data we see that on the Pentium and the Alpha accuracy in the stack and globals is most important. Accuracy in globals improves results for more benchmarks than accuracy in the stack. On the SPARC, accuracy in the heap is also quite important.

4.3 Usefulness of accuracy in finding memory leaks

The previous section demonstrated that garbage collection with accuracy frequently reclaims more objects than conservative garbage collection. Thus, a leak detector based on accurate garbage collection will likely find more memory leaks than a leak detector based on conservative garbage collection.

Even for our suite of benchmarks, we detected four programs where accurate garbage collection reclaimed more objects than explicit deallocation: *bc*, *ft*, *yacr2*, *ijpeg*. Only one of these leaks (*ft*) shows up in Table 6, since Table 6 presents the *average* difference between explicit deallocation and garbage collection. In our experiments, conservative garbage collection reclaimed more objects than explicit deallocation in only one program *ft*. All these

	Alpha		Pentium		SPARC	
Benchmark	avg. $\frac{gc_{shg}-free}{free}$	frac. gc	avg. $\frac{gc_{shg}-free}{free}$	frac. gc	avg. $\frac{gc_{shg}-free}{free}$	frac. gc
anagram-5	0	0	0	0	0.00	0.50
anagram-50	0	0	0	0	0.00	0.50
ks-5	0	0	0	0	0	0
ks-50	0	0	0	0	0	0
bc-5	0.00	0.83	0.00	0.83	0.01	0.83
bc-50	0.00	0.98	0.01	0.98	0.03	0.98
ft-5	-0.00	0.83	0	0	0.01	0.83
ft-50	0.00	0.98	0	0	0.00	0.96
yacr2-5	0.00	0.75	0.02	0.75	0.02	0.75
yacr2-50	0.00	0.35	0.02	0.95	0.00	0.95
gzip-5	0.23	0.67	0.22	0.67	0.22	0.67
gzip-50	0.57	0.82	0.57	0.82	0.26	0.82
ijpeg-5	0	0	0.10	0.80	0.16	0.20
ijpeg-50	0.07	0.33	0.32	0.98	0.05	0.33

Table 6: Accurate gc versus explicit deallocation

	Alpha		Pentium		SPARC	
Benchmark	avg. $\frac{gc-free}{free}$	frac. gc	avg. $\frac{gc-free}{free}$	frac. gc	avg. $\frac{gc-free}{free}$	frac. gc
anagram-5	0	0	0	0	0.00	0.50
anagram-50	0	0	0	0	0.00	0.50
ks-5	0	0	0	0	0	0
ks-50	0	0	0	0	0	0
bc-5	0.01	0.83	0.01	0.83	0.04	0.83
bc-50	0.00	0.98	0.01	0.98	0.04	0.98
ft-5	-0.00	0.83	0.01	0.33	0.02	0.83
ft-50	0.00	0.98	0.01	0.33	0.00	0.98
yacr2-5	0.02	0.75	0.05	0.75	0.04	0.75
yacr2-50	0.03	0.35	0.06	0.95	0.05	0.95
gzip-5	0.23	0.67	0.22	0.67	0.65	0.67
gzip-50	0.57	0.82	0.66	0.82	1.08	0.82
ijpeg-5	0.07	0.20	0.97	0.80	0.93	0.80
ijpeg-50	0.07	0.33	0.78	0.98	0.67	0.98

Table 7: Conservative gc versus explicit deallocation

	Alpha		Pentium		SPARC	
Benchmark	avg. $\frac{gc-gc_{shg}}{gc}$	frac. gc	avg. $\frac{gc-gc_{shg}}{gc}$	frac. gc	avg. $\frac{gc-gc_{shg}}{gc}$	frac. gc
gctest3-5	0	0	(g) 0.00	0.67	0	0
gctest3-50	0	0	(g) 0.01	0.92	0	0
gctest-5	0	0	0	0	0	0
gctest-50	0	0	0	0	0	0
anagram-5	0	0	0	0	0	0
anagram-50	0	0	0	0	0	0
ks-5	0	0	0	0	0	0
ks-50	0	0	0	0	0	0
bc-5	(g) 0.00	0.83	(sg) 0.01	0.83	(shg) 0.03	0.83
bc-50	(g) 0.00	0.98	(sg) 0.00	0.98	(shg) 0.01	0.98
ft-5	0	0	(g) 0.01	0.33	(shg) 0.01	0.17
ft-50	0	0	(g) 0.01	0.33	(shg) 0.00	0.61
yacr2-5	(g) 0.02	0.25	(g) 0.03	0.25	(g) 0.02	0.25
yacr2-50	(g) 0.02	0.35	(g) 0.03	0.37	(g) 0.04	0.37
bshift-5	0	0	(g) 0.05	0.83	(shg) 0.10	0.67
bshift-50	(g) 0.02	0.05	(g) 0.11	0.77	(shg) 0.25	0.93
li-5	0	0	(g) 0.02	0.67	(shg) 0.00	0.83
li-50	0	0	(g) -0.00	0.94	(shg) 0.01	0.98
gzip-5	0	0	0	0	(s) 0.16	0.33
gzip-50	0	0	(h) 0.02	0.09	(s) 0.27	0.55
ijpeg-5	(s) 0.05	0.20	(sg) 0.39	0.80	(sg) 0.35	0.80
ijpeg-50	0	0	(sg) 0.24	0.96	(shg) 0.35	0.98

Table 8: Accurate versus conservative gc

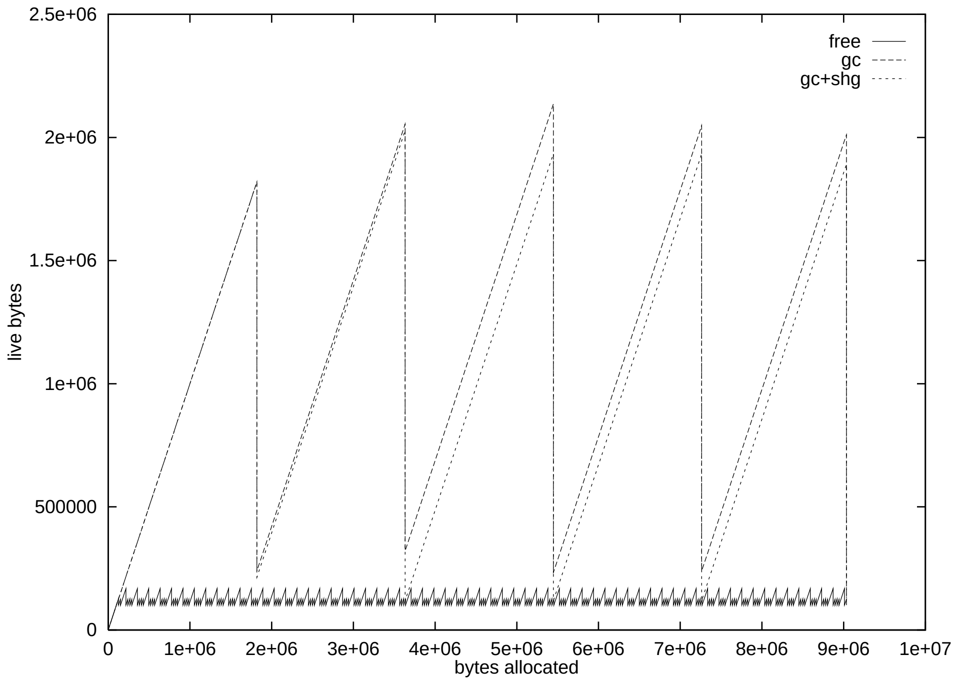

Figure 2: Memory usage of *ijpeg-5 on a SPARC*

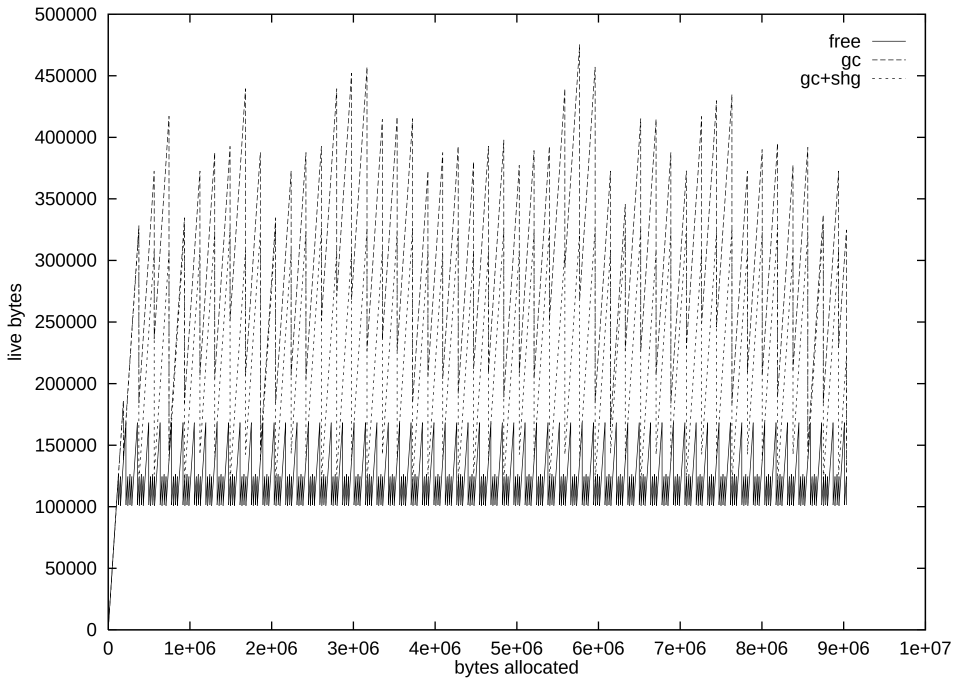

Figure 3: Memory usage of *ijpeg-50 on a SPARC*

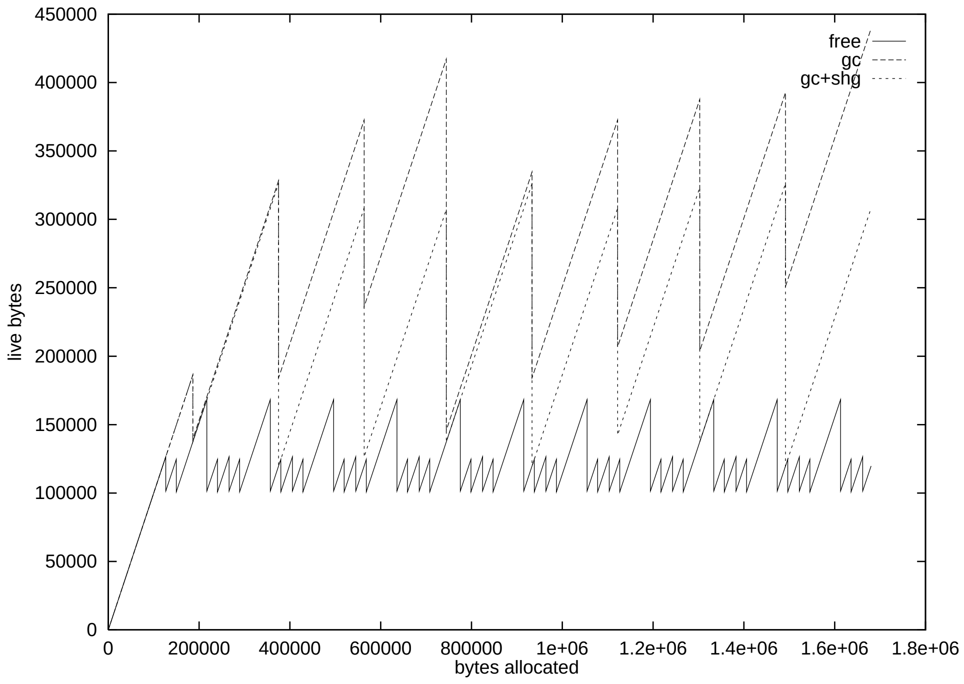

Figure 4: Memory usage of *ijpeg-50 on a SPARC* (first 9 collections)

potential leaks are small (typically 100 bytes or less) and are not significant. However, it is important to keep in mind that the benchmarks we used are well-established and well-studied programs; thus it would have been surprising to find significant leaks in them.

Finally, our methodology was effective in indirectly finding two leaks in the BDW collector[4]. The first of the two leaks was a bug in the BDW collector; we fixed it before taking our measurements. The second leak turned out to be a known inaccuracy that caused the BDW collector to leak memory but run faster. We suspected these leaks when we noticed discrepancies between data for different levels of accuracy.

4.4 Explanation of results

The different performance of conservative garbage collection on different hardware platforms could be due to two reasons. First, different memory layouts on different hardware platforms could cause different objects to be conservatively retained by the conservative collector. Second, different compiler optimizations on different hardware platforms may create different values in memory which may cause different objects to be retained by the conservative collector.

To better understand the reason for the differences across platforms we reran our experiments on the SPARC but this time using a different starting address for the heap: we forced the heap to start at 0xbb000, which is much lower than the default start address (see Table 1). If memory layout determines the behavior of conservative collection then we should see a significant difference between results for the two heap layouts.

[4]We confirmed these with Hans Boehm via email in April and May 2000.

Benchmark	Original Average $\frac{gc-gc_{shg}}{gc}$	Lower heap layout Average $\frac{gc-gc_{shg}}{gc}$
gctest3-5	0	0
gctest-5	0	(sg) 0.39
anagram-5	0	0
ks-5	0	0
bc-5	(shg) 0.03	(shg) 0.39
ft-5	(shg) 0.01	(shg) 0.00
yacr2-5	(g) 0.02	0
bshift-5	(shg) 0.10	(hg) 0.04
li-5	(shg) 0.00	(s) 0.00
gzip-5	(s) 0.16	(s) 0.16
ijpeg-5	(sg) 0.35	(sg) 0.21

Table 9: Effect of changing heap layout for SPARC

Table 9 presents the results of this experiment. The “Original” column of Table 9 is identical to the *Average* $\frac{gc-gc_{shg}}{gc}$ column in Table 8 for the SPARC except that it presents numbers only for the 5 garbage collection runs. The “Lower heap layout” column is computed similarly to the “Original” column except that it uses a lower starting address for the heap. From this table we see that the starting address of the heap impacts the effectiveness of conservative garbage collection. However, there is no clear pattern. Lowering the heap layout reduces the effectiveness of conservative garbage collection for two of the benchmarks (*gctest* and *bc*) and increases the effectiveness for four of the benchmarks (*ft*, *yacr2*, *bshift*, and *ijpeg*).

These results suggest that the variation in performance of conservative garbage collection across platforms is largely due to the different memory layout. These results also suggest that the best way to use existing leak detectors is to run them several times, each time with a different heap layout. Since different heap layouts will find different leaks, the end result will be that more leaks will be detected.

4.5 Summary of results

Our results show that the hardware platform significantly affects the performance of conservative garbage collection algorithms. Particularly, we found that on the Alpha, accuracy did not make a significant difference in the garbage collector's ability to reclaim objects. Moreover, on the Alpha, a 64-bit hardware platform, all our garbage collectors were almost as effective in reclaiming objects as explicit deallocation.

On the two 32-bit platforms, accuracy in garbage collection is much more important for reclaiming objects. For the Pentium and particularly the SPARC, accuracy significantly affects a garbage collector's ability to reclaim objects.

Finally, gc_{shg} (our most accurate collector) is much more effective in reclaiming objects than conservative garbage collection on the 32-bit platforms. Thus on the 32-bit hardware platforms, we expect that a leak detector based on gc_{shg} will find more memory leaks than existing leak detectors such as Purify [9] which are based on conservative garbage collection.

5. RELATED WORK

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

Bartlett [3], Zorn [18], Smith and Morrisett [13], and Agesen *et al.* [2] compare different garbage collection alternatives with respect to memory consumption. Bartlett [3] describes versions of his mostly-copying garbage collector that differ in stack accuracy. Zorn [18] compares the Boehm-Demers-Weiser collector to a number of explicit memory management implementations. Smith and Morrisett [13] 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 live 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 liveness on the number of live bytes after an accurate garbage collection. We do not consider liveness information at this point but believe that liveness is worth considering.

Zorn [18], Smith and Morrisett [13], and Hicks *et al.* [10] compare different memory management schemes with respect to their efficiency. Zorn [17] 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.

Diwan *et al.* [8], Agesen *et al.* [2], and Stichnoth *et al.* look at how to perform accurate garbage collection in compiled type-safe languages. Diwan *et al.* [8] describe how the compiler and run-time system of Modula-3 can support accurate garbage collection. Agesen *et al.* [2] and Stichnoth *et al.* 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 methodology allows us to have type accuracy even for C and without compiler support. This comes at the cost of having to run the program twice with the same input, but it is still useful as a leak detector and, of course, as a study in garbage collector accuracy.

In work concurrent to ours, Shaman *et al.* [12] 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.

Hastings and Joyce [9], Dion and Monier [7], and GreatCircle [1] describe leak detectors based on the Boehm-Demers-Weiser collector [6]. The Boehm-Demers-Weiser collector can also be used as a leak detector [5]. Our scheme uses more accurate information than these detectors and is thus capable of finding more leaks in programs.

6. FUTURE WORK

The work presented here is a preliminary investigation of the usefulness of accuracy and the potential usefulness of our tools. We are continuing work in this area in several directions:

- Evaluating the effectiveness of richer forms of accuracy, for example where accuracy involves liveness information. Prior work, with the exception of a limit study [12], has investigated only intraprocedural liveness of local variables [2]. We are investigating the usefulness of interprocedural liveness information.

- Language-independent accurate garbage collection. While our current run-time analysis allows us to do language-independent leak detection, we cannot yet use it for language-independent accurate garbage collection since it requires two runs. We are modifying our analysis so that it requires only a single run and are investigating compiler support to reduce the overhead of the run-time type analysis.

- Leak detection. As we demonstrated, our analysis is more accurate and thus more effective at finding leaks than existing leak detectors [5, 9, 1, 7]. However, our method requires two runs of the program, which is infeasible especially for interactive programs. We are modifying our analysis so that it requires only a single run.

- Improving conservative garbage collectors on "unfavorable" platforms. With the BDW collector [5] one can "blacklist" ranges of memory. The memory allocator avoids allocating objects in the black-listed memory. Similarly on some linker/loaders, one can specify the starting address for the data area. We are investigating using one or both of these possibilities to improve the performance of conservative garbage collection on unfavorable platforms, such as SPARC/Solaris.

7. CONCLUSIONS

This paper describes a novel approach to obtaining type-accurate information in a language and hardware independent manner. While this technique is not yet suitable to use for garbage collection, we show that this approach is more effective at finding leaks in C programs than prior approaches that use conservative garbage collection.

We used our approach to compare garbage collectors with different levels of accuracy and explicit deallocation on three hardware platforms, Alpha, Pentium, and SPARC. We found that the choice of hardware platform greatly determines where accuracy is needed. In particular, on some hardware platforms, such as the 64-bit Alpha, accuracy is not important for reclaiming objects. On other hardware platforms such as the SPARC accuracy in all regions of memory is important for reclaiming objects.

8. ACKNOWLEDGMENTS

We would like to thank Dirk Grunwald, Tony Hosking, and Alex Wolf for valuable discussions during this project. We would also like to thank Martin Burtscher, Rick Hudson, Eliot Moss, and Darko Stefanovic for comments on a draft of this paper. We would like to thank Hans Boehm for helping us understand the Boehm-Demers-Weiser collector.

9. REFERENCES

[1] Great Circle – Real-time error detection and code diagnosis for developers. www.geodesic.com/solutions/greatcircle.html.

[2] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 269–279, Montreal, Canada, June 1998.

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

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

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

[6] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and experience*, 1988.

[7] Jeremy Dion and Louis Monier. Third degree. research.compaq.com /wrl/projects/om/third.html.

[8] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. In *PLDI'92*, pages 273–283, July 1992.

[9] 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.

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

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

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

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

[14] James Stichnoth, Guei-Yuan Lueh, and Michał Cuerniak. Support for garbage collection at every instruction in a Java compiler. In *PLDI'99*, pages 118–127, May 1999.

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

[16] 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.

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

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