Effectiveness of Garbage Collection and Explicit Deallocation

by

Martin Hirzel

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

2000

This thesis entitled:
Effectiveness of Garbage Collection and Explicit Deallocation
written by Martin Hirzel
has been approved for the Department of Computer Science

Amer Diwan

Dirk Grunwald

Alexander Wolf

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Hirzel, Martin (M.S., Computer Science)

Effectiveness of Garbage Collection and Explicit Deallocation

Thesis directed by Prof. Amer Diwan

This work compares the effectiveness of accurate garbage collection, conservative garbage collection, and explicit allocation/deallocation for C programs. Traditionally, garbage collectors for C are conservative, they can only recognize non-pointers if those are not aligned or point into a memory range known not to contain live objects. This means that they may fail to reclaim memory, even though it is only reachable through non-pointers. We call exact pointer/non-pointer information accuracy. Adding accuracy to a conservative garbage collector means enabling it to be less conservative for some pointers.

We do a runtime analysis on C-programs that finds, for each global object, heap object, and stack frame type, which offsets contain pointers for a particular run. In a second run with the same inputs, we make this accuracy information available to the Boehm-Demers-Weiser (BDW) garbage collector. This enables BDW to ignore non-pointer values in the mark phase, even if they look like pointers to objects maintained by BDW, and thus to potentially reclaim more objects in the sweep phase. This approach gives us the effect of accurate garbage collection, it shows how much memory the original BDW actually fails to reclaim due to its conservatism.

We measure the effectiveness of BDW for some realistic C benchmarks. It can independently use or ignore the accuracy information for stack, heap, and globals, and it can allow or disallow the explicit deallocation present in the benchmarks. Furthermore, by logging statistics about *malloc()* and *free()*, we also measure how effective explicit deallocation is in reclaiming memory if the garbage collector is not active. We compare how much memory gets reclaimed at each garbage collection for the whole spectrum of

accuracy and for explicit deallocation.

For eight of our twelve benchmarks, there were no differences in effectiveness for explicit deallocation or any of the accurate garbage collector configurations. In those benchmarks where there were differences, we could see almost no pattern. Often, it was one single aspect that was responsible for a better effectiveness, but this aspect varied widely from benchmark to benchmark and even from garbage collection to garbage collection. Aside from the sed benchmark, where some schemes reclaimed 100% of the storage, the most extreme differences in number of live bytes in objects was 25.53%. This means that all techniques are in the same ball-park for effectiveness.

To my parents.

Acknowledgements

I am grateful to Amer Diwan for all the good ideas, motivation, encouragement and time he gave me. He is a great advisor, and has taught me a lot in these past months. The initial idea for this research and the code for the runtime analysis which I describe in chapter 2 come from him.

Thanks also to Hans Boehm, who confirmed my findings about his garbage collector.

I am grateful to the German Academic Exchange Service, which enabled me to come to Boulder, and to the German National Scholarship Foundation.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

1.1 Definitions

The traditional heap storage management for C is *explicit deallocation*, where the programmer has to call *free()* to give memory back to the runtime system. An alternative to this, *garbage collection (gc)*, automatically reclaims memory when it cannot be reached anymore (we call unreachable objects *dead*). The research for this thesis was conducted using the Boehm-Demers-Weiser (*BDW*) garbage collector [7, 8, 9], which is essentially a *mark-and-sweep* garbage collector. First, the mark phase marks all live objects: it starts from a *root set* (stack, globals, and registers), and builds the transitive closure of reachable objects by following pointers. Second, the *sweep* phase reclaims everything that has not been marked.

In the mark phase, the garbage collector must make safe assumptions about which memory locations contain pointers that must be followed to mark reachable objects as live. In type-safe languages, it can reliably distinguish pointers from non-pointers, it has *accuracy*. This information is not available in C, so the garbage collector must be *conservative*: if a value can be interpreted as a pointer, it must be treated as a pointer, even if it is declared to be something different.

The basic idea for garbage collection follows a *stop-the-world* model: the program execution gets stopped until the garbage collection is over. Since this may cause problems for real-time applications and even inconvenience the user, *incremental* garbage

collectors have been invented. Those perform only a small amount of work when the garbage collection gets triggered. Then they allow the program to run on, doing a little work once in a while towards finishing the started garbage collection.

In many high-level programming languages, the language definition assumes the presence of a garbage collector, and often the programmer cannot even choose to deallocate memory explicitly. This has gained acceptance with the wide-spread use of Java. Explicit deallocation is not trivial, and errors in doing so may lead to bugs that are difficult to find or even to reproduce. Furthermore, if the programmer has to keep explicit deallocation in mind, this may distract from the problem at hand and in the worst case force an inferior programming style to avoid the complexities of inserting the calls to *free()* in all the right places. As Wilson et al. [25] write: this can "incur major costs in productivity and, to put it plainly, human costs in sheer frustration, anxiety, and general suffering".

1.2 Garbage collection versus explicit deallocation

There are surprisingly few studies that compare garbage collection to explicit deallocation. Zorn [27] compares the Boehm-Demers-Weiser conservative garbage collector to explicit memory management for C. He finds that the CPU performance is almost the same. From our experience, we can add that since BDW is incremental, there are no stop times that can be observed by the user.

On the other hand, Zorn found the heap size of the program to be often twice as big for garbage collection as for explicit deallocation. There are three reasons for this: fragmentation, frequency of garbage collection, and effectiveness of garbage collection. Zorn finds that the fragmentation is comparable to that of the explicit memory allocators he studied. He states that increasing the frequency of garbage collection does reduce the heap size, but not always significantly. The third question, how well BDW can find dead objects and therefore how effective it is in reclaiming memory, is left open. This

thesis helps to answer it.

It is also possible to run BDW, but do explicit deallocation where this is easy for the programmer. The programmer may be able to free objects that BDW did not find to be dead, and BDW may be able to free objects where the programmer did not bother to track liveness. On the other hand, the effectiveness of this combined effort might be no significant improvement over either one of them. We will present experimental results and discuss this approach.

1.3 Conservative versus accurate garbage collection

Conservative garbage collectors might fail to reclaim dead objects because there are non-pointer values in live objects that can be interpreted as pointers to them. Consider for example the C program in figure 1.1. The first **for**-loop fills the array a with values that are of the same order of magnitude as heap addresses. The second **for**-loop allocates heap objects which become dead immediately, since they are not reachable by traversing pointers from local or global variables. Yet, some of the elements of a happen to look like pointers to objects created by f, so the conservative collector does not reclaim them.

If this happens too often, the effectiveness of the garbage collector will suffer significantly and the heap size will be much larger than it needs to be. This may lead to poor cache or even paging behavior. Also, the performance of the garbage collector depends on how many objects it has to scan, and a more accurate collector has to scan less objects. The paper describing Barlett's mostly-copying garbage collector [4] compares different accuracy levels for the stack, and finds little difference for the two measured benchmarks. Boehm and Weiser [9] state that conservativism had little effect on the effectiveness of an predecessor of BDW, but they do not compare it to an actual accurate garbage collector. We will discuss these and more articles in chapter 7.

So far, there are no accurate garbage collectors for C. One can, however, imagine

```
#include ⟨stdio.h⟩
#include ⟨stdlib.h⟩
#include "gc.h"

void f(int i){
  char*p ← malloc(sizeof(long)*250);
  *p ← i;
}

#define ASIZE 500

int main(int argc, char*argv[]){
  unsigned long a[ASIZE];
  unsigned long v;
  int i;
  init(&argc, argv);
  v ← 1 4000 0000₁₆;
  for(i ← 0; i <ASIZE; i++){
    a[i] ← v;
    v+← 2 000;
  }
  for(i ← 0; i < 200; i++)
    f(i);
  GC_gcollect();
  return 0;
}
```

Figure 1.1: `demo.c`. For this C program, the conservative collector fails to reclaim some objects.

techniques to obtain accuracy, if not for all (global, heap, stack) data, then at least for some of it. For example, in figure 1.1, the values in the stack-allocated array a never get used as pointers, even though the weak typing of C would permit it. Providing accuracy information would make BDW strictly more effective, but would also come at some cost. Techniques that can be used as a starting point here are described in [1, 22, 12]. On the other hand, it is not clear how much room for improvement BDW leaves for such efforts. We independently simulate accuracy for heap, stack, and globals, and can thus compare BDW to hypothetical accurate garbage collectors for C in terms of effectiveness.

1.4 Comparing the effectiveness

Figure 1.2 shows the effectiveness of garbage collection for the example program from figure 1.1. A total of 408 000 bytes get allocated over the run of the program (the object size **sizeof**(**long**)$*$250 gets rounded up by *malloc()*). The solid/dashed line shows the live size if we run an accurate/conservative garbage collector, respectively. Garbage collections are triggered after every 100 000 bytes of allocation. The accurate collector is able to reclaim all heap objects every time. The conservative collector misses some, causing the live size to be greater than that for the conservative collector. Therefore, the total number of live bytes in objects never exceeds 100 000 for the accurate collector, while it goes up to 108 120 bytes with the conservative collector, which is 8.12% worse.

We use the number of bytes in live objects as our basic metric for the effectiveness of garbage collectors. We do measurements like the one shown in figure 1.2 for a variety of C programs, for a variety of gc accuracy levels, and with or without allowing explicit deallocation. From the results, we get a feeling for how often a situation like in figure 1.1 can be found in practice. We can also identify the culprit of the inaccuracy: for which memory area would we need accurate information for the gc to be effective? Furthermore, we see whether explicit deallocation is more effective, less effective, or a

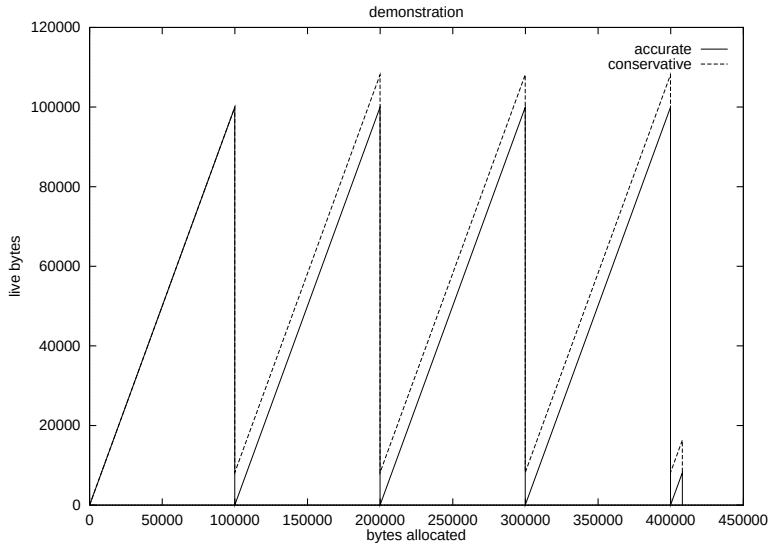


Figure 1.2: Results for `demo.c`. The x axis shows the total number of bytes allocated, the y axis shows how many of them are still live.

reasonable addition to garbage collection. The effectiveness of all these variations are compared on an equal footing, abstracting from those algorithmic details of the garbage collector that do not affect its pointer/non-pointer distinction.

We obtain our accurate information from a dynamic pointer analysis. Chapter 2 describes the details of the analysis and also discusses how accurate it really is. One fundamental point is that it happens at run-time. That means that it provides an upper bound on the precision of static analyses of otherwise equal strength. If a location never contains a pointer in one particular run, it may still end up as a pointer in a second run with different inputs.

1.5 Overview of this work

We start by describing the runtime analysis that collects accurate information in chapter 2. Since we want to supply this information to BDW, we describe the relevant details of that garbage collector in chapter 3. Given this background knowledge, we understand how BDW gets accurate in chapter 4.

Chapter 5 describes the benchmarks we used for our experiments. In chapter 6, we have a close look at the measurements we took. It provides numbers and graphs gathered on effectiveness for the different accuracy configurations and explicit deallocation.

We conclude by reviewing the related work in chapter 7, and by giving a summary, reflection, and ideas for future work in chapter 8.

Chapter 2

Run-time Analysis

2.1 Two program runs

We want to provide the Boehm/Demers/Weiser garbage collector with accurate information about which values are pointers and which are not. To do this, we run the benchmark twice with the same input. On the first run, we keep track of which locations hold pointers. When the first run terminates, we write this information out to disk. On the second run, we read it in again. Using this, when BDW is about to mark an object from a non-pointer, we can stop it from doing so.

2.2 Instrumenting the first run

On the first run, we keep track of which locations hold pointers. To do this, we insert calls to book-keeping code whenever the original program does something that affects this information. We use the term *instrumentation* both for this book-keeping code, and for the process of inserting it. The latter is done using SUIF [23]. SUIF stands for Stanford University Intermediate Format, it is a research compiler infrastructure. It enables you to write a compiler pass that can manipulate the abstract syntax tree representation of programs through a convenient C++ API.

The instrumentation proceeds as follows.

(1) The input is the set of C files of the benchmark.

(2) Compile this to an intermediate format, which can be read in by our pass.

(3) Insert calls to and prototypes for instrumentation functions.

(4) Compile back to C.

(5) Compile and link with the actual instrumentation functions' implementations, which is written in C++. For this step, we use `gcc`.

For each user object, we maintain information of which offsets inside the user object contain pointers. This information is kept in a *model* for the user object, that is, a C++ object that has a bit vector for recording where the pointers are. For this purpose, we distinguish four different kinds of user objects. *Heap objects* are instances of heap allocated data structures. *Global objects* are the contents of global variables. *Stack frame instances* are basically the space on the call stack for the formal and local variables of a procedure invocation. Finally, we also have models for *stack frame types*: if any stack frame instance has a pointer at offset o, we set the corresponding bit for the stack frame type. The instrumentation makes sure these models get correctly initialized and maintained through the run of the program.

2.3 Analyzing the first run

2.3.1 High-level view

Pointer/non-pointer information is propagated for assignments, argument passing and return values. Consider an assignment $r \leftarrow s$, where r is the recipient and s is the source. If r has so far been considered a non-pointer, but s is a pointer, then we record that the location of r can hold a pointer as well. We distinguish between two cases for the source s. If s is an allocation or address-taking statement, then we statically know that r will become a pointer. Otherwise, we look up the pointer/non-pointer information gathered for the operands of the source s. If any of them can hold pointers, we record

that the recipient r can hold a pointer, otherwise, we leave the pointer/non-pointer information for r untouched. In other words, pointerness gets inserted at *malloc()*s and &-statements, and gets propagated as values get written to locations in the dynamic execution of the program.

2.3.2 Details

Figure 2.1 shows a tiny C program before and after instrumentation. The rest of this section describes the implementation of the instrumentation functions. This level of detail is not crucial for understanding the thesis; the reader may want to skip the remaining subsections and continue with the discussion in section 2.4.

2.3.3 Program startup

Call *process_global()* to create models for global objects.

2.3.4 Call, caller side

Remember the address where the return value will get stored with *start_call()*. Record which actuals contain pointers with *process_ptr_arg_source()* and *process_arg_source()*.

2.3.5 Call, callee side

Create a model for the stack frame instance with *enter_proc()*. If actual was a pointer, let formal be a pointer with *process_formal()*. Figure out the size of the frame with *note_stack_alloc()* and *end_stack_alloc()*.

2.3.6 Return

Before the call, we remembered the address of where the return value will get stored. Now, with *note_return_ptr_source()* or *note_return_source()*, possibly note that

Before instrumentation.

```
long g ← 42;

long*f(long**x){
  *x ← &g;
  return malloc(sizeof(long));
}

int main(){
  long*p, *q;
  p ← f(&q);
  return 0;
}
```

After instrumentation.

```
long g ← 42l;

extern long *f(long **x){
  void *suif_tmp;

  enter_proc(0u);
  process_formal(0u, &x);
  note_stack_alloc(8u, &x);
  note_stack_alloc(8u, &suif_tmp);
  end_stack_alloc();
  note_assignment_target(x);
  note_assignment_ptr_source();
  *x ← &g;
  note_assignment_target(&suif_tmp);
  note_assignment_ptr_source();
  suif_tmp ← my_malloc(8ul);
  note_return_source(&suif_tmp);
  end_return();
  return (long *)suif_tmp;
}

extern int main(){
  long *p;
  long *q;

  enter_proc(1u);
  note_global(8u, &g);
  note_stack_alloc(8u, &p);
  note_stack_alloc(8u, &q);
  end_stack_alloc();
  start_call(1u, &p);
  process_ptr_arg_source(0u);
  p ← f(&q);
  end_program();
  return 0;
}
```

Figure 2.1: Instrumented code. For simplicity, we omitted `#includes` and function prototypes.

that location becomes a pointer. Get rid of the stack frame instance model with *end_return()*; we stored the accuracy information in the stack frame type model. A special case is a "return" via *longjmp()*, in which case we must tidy up the stack model by a *note_after_setjmp()* in the caller.

2.3.7 Malloc and friends

This instrumentation does not get inserted immediately into the user code. Instead, we have wrappers for *malloc(), calloc(), realloc(),* and *free()* that call the respective instrumentation functions *note_allocation()*, *note_reallocation()*, and *note_deallocation()*. Those create and update the models for heap objects.

2.3.8 Assignment

First, we remember the address of the location where the assigned value will be stored with *note_assignment_target()*. Then, we propagate the pointerness to that with *note_assignment_source()* or *note_assignment_ptr_source()*

2.3.9 Program termination

At return from *main()* or *exit()* from other procedures, *end_program()* makes sure that all the gathered information is written to file.

2.4 Discussion

2.4.1 Conservativism

One can say that this analysis corresponds to a *flow-insensitive* static analysis. If a location holds a pointer at any time during program execution, we say it can hold a pointer at all times. This corresponds to type analyses, which are usually flow-insensitive. One can also say that this analysis corresponds to a *context-insensitive*

static analysis. If a stack-allocated variable holds a pointer in any invocation of its procedure, we say it can hold a pointer at every invocation of the procedure.

One more source of inaccuracy is the treatment of arithmetic. If any operand of an arithmetic expression is a pointer, we say that the result of the expression is a pointer. This is usually true for expressions like $p \leftarrow q + 1$, but not for expressions like $n \leftarrow p - q$ or $b \leftarrow (p \leq q)$.

2.4.2 Precision

This analysis is more precise than static pointer analyses in that it keeps all heap objects separate. A static pointer analysis computes a *points-to graph* where vertices model memory locations and edges model possible points-to relationships. Since the number of heap objects a program can allocate is theoretically infinite, a static analysis must bound the number of corresponding vertices in some way. Techniques for this range from one vertex for all heap objects [13] over one vertex per allocation site [17] to shape-oriented techniques [16], but our analysis is strictly more precise than any of these in this respect.

Since this analysis only considers a location to hold a pointer if it seemed so in one particular run, it is more precise for this run than a static analysis could be. It may be even more powerful than flow- and context-sensitive analyses.

2.4.3 Conclusion

We believe that even given the conservative aspects of our analysis, it does in practice give us an upper bound on how accurate you can make a garbage collector for C. This is based on the intuition that for real C programs, the imprecisions we described above will either not appear in the first place, or be caught by the pointer/non-pointer distinction techniques of BDW on top of which we build our accuracy. But this remains, of course, material for further research.

Chapter 3

Boehm-Demers-Weiser Garbage Collector

3.1 Classification

The Boehm-Demers-Weiser garbage collector (*BDW* for short) is a conservative garbage collector for C and C++. Boehm and Weiser describe a predecessor of BDW in [9]. In [8], Boehm, Demers and Shenker describe how you can make a mark-and-sweep garbage collector mostly incremental and parallel. Their technique is implemented in the BDW garbage collector, which is apparent in several design decisions throughout the source code.

BDW has been carefully tuned. Already in 1993, Zorn [27] finds that it is about as fast as explicit memory managers. This is achieved mainly by carefully keeping in mind which parts of the memory hierarchy are touched throughout the algorithm. Also, there are macros, unrolled loops, alternative code paths and other hand-optimizations.

The Boehm-Demers-Weiser garbage collector supports a lot of platforms. Guided by `#ifdefs`, it can be compiled for a number of different machines, operating systems, and thread packages. We use it only on the Alpha running Digital Unix and in a single-threaded environment. We allow *interior pointers*, that is, pointers do not have to point to the head of heap objects only. We enable BDW's debugging book-keeping, since we use some of it for our statistics. We allow BDW to round object sizes (see below).

3.2 Pointer/non-pointer distinction

The memory managed by BDW is partitioned into *heap blocks*, which for our purpose are $2^{13} = 8\,192$ Bytes large. Objects are considered *large* when they do not fit into one heap block, and are treated specially. A small object heap block contains only objects of one size, so objects are segregated by size classes. Not every real small object size has its own size class; instead, sizes are rounded up to predetermined size classes.

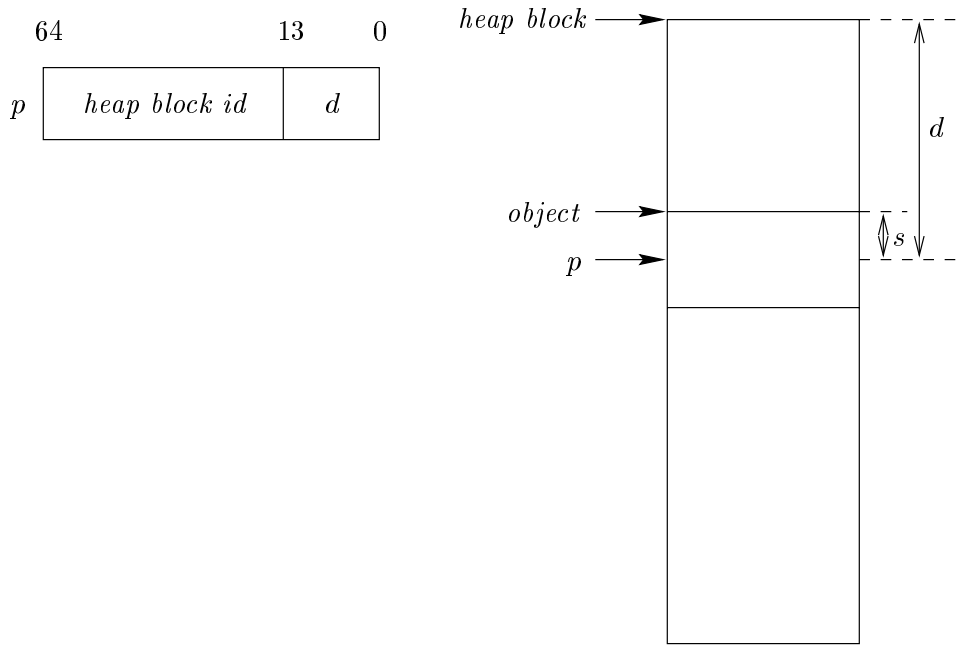


Figure 3.1: Boehm-Demers-Weiser Pointer

Figure 3.1 shows a 64-bit Alpha-pointer p, and how it is used to access various BDW data structures. Its 51 higher order bits are used as a heap block id. The heap block header, which is stored separately from the heap block itself, is looked up in an index structure (not shown). The header contains a map that, given the displacement d of p in its heap block, returns the displacement s of p in its object. This way, we get the pointer to the head of an object from an interior pointer.

BDW distinguishes pointers from non-pointers before and in the macro *PUSH_CONTENTS()* in the file `gc_mark.h`. The paper [9] by Boehm and Weiser de-

scribes this logic, the only difference is that in our case, interior pointers are allowed. A value p is considered a pointer if and only if

- it is aligned to an 8 Byte word boundary
- $least\_ha \leq p \leq greatest\_ha$, where *least_ha* and *greatest_ha* are rough estimates of the least and greatest plausible heap address, respectively
- there is a heap block header for that heap block
- the heap block contained at least one live object after the last garbage collection

3.3 Mark stack

As mentioned above, BDW is incremental: it does not have to do a whole collection all at once, but can do one small amount of work at a time. For the mark phase, this is achieved by having a *mark stack* of memory areas to scan for pointers. For our purpose, a *mark stack entry* consists of a pointer to the start of a memory area, and of the length of that area. To initiate a garbage collection, BDW pushes mark stack entries for the roots, that is, the current stack and globals, on the mark stack. Later, it repeatedly pops a mark stack entry and looks at each aligned value v in the memory described by it. If it determines that v is a pointer, it looks whether the object o pointed to by v is already marked. If not so, it marks it and creates a new mark stack entry for the memory area o.

The mark phase is complete when the mark stack is empty. Now, the sweep phase starts, where everything not marked during the mark phase gets reclaimed. The details of the sweep phase are not relevant for this thesis.

Chapter 4

Using the Accuracy

4.1 High-level view

Chapter 2 describes how we collect accurate pointer/non-pointer information during a first run of the program. In the second run, we want to provide this information to BDW to simulate an accurate garbage collector. How this is done differs slightly depending on the kind of location (stack, heap, or pointer) we provide accurate pointer/non-pointer information for.

The second run of a benchmark understands the command-line argument `-a` which specifies what kind of accuracy to use and what kind to ignore. For example, `-astack,glob` means that the collector has accurate information about which stack and global locations contain pointers, but it must be conservative for locations in heap objects. Let p be a value encountered during the mark phase. To decide whether to mark the object it points to and scan it for more pointers, we evaluate the predicate *is-pointer(p)*.

$$\textit{is-pointer}(p) \equiv \Big(\neg \textit{use-accuracy(k)} \vee \textit{is-pointer}_{\text{analysis},k}(p)\Big) \wedge \textit{is-pointer}_{\text{BDW}}(p)$$

We first have to figure out which kind $k \in \{\textit{Stack,Heap,Global}\}$ of location p resides in. If we do not use the accuracy for that kind of location, we just move on to BDW's own logic for $\textit{is-pointer}_{\text{BDW}}(p)$. Otherwise we look up the information $\textit{is-pointer}_{\text{analysis},k}(p)$ gathered by the analysis in the first run, and intercept BDW from

even considering the candidate pointer if that yields **false**. In the case where we fall through to BDW's pointer/non-pointer distinction mechanism, $\mathit{is\text{-}pointer}_{\mathrm{BDW}}(p)$ gets evaluated as described in section 3.2. Once p has passed all checks, if the target object o is not yet marked, BDW marks o and and pushes a mark stack entry describing o on the mark stack, so its area will eventually be scanned for pointers.

4.2 Using accuracy for the heap

Our wrapper *my_malloc* allocates one word more than requested by the client, and uses this to store an id number. Since our benchmarks are deterministic and use the same inputs on both runs, corresponding heap objects get the same id number. When BDW works off the mark stack, it looks at the memory area described by a mark stack entry word for word. Let p be a pointer and s be its source, that is, the heap memory location containing p. We find the start of the heap object from the interior pointer s using BDW's data structures (see figure 3.1). This gives us the heap object id and the offset, which is all we need to access the information from the run-time analysis.

4.3 Using accuracy for the stack

The user stack of procedure activation records is just one large entry of the mark stack of memory areas to scan for pointers. This entry is processed in exactly the same manner as entries corresponding to other kinds of memory. In and of itself, it gives no clues as to where individual stack frame instances start or end. To avoid changing the incremental design of BDW too much, we had to device a way of finding out what stack frame type a given stack location belonged to, and what its offset from the frame pointer was.

For this purpose, like for the first run, we also instrument the benchmark for the second run, only that we bind it to a different set of instrumentation functions. With other words, the C code for the second run looks exactly like the right side of figure 2.1,

but the implementation of *enter_proc, process_formal, note_stack_alloc etc.* is different.

Most of the instrumentation functions for the second run just have empty bodies. With *enter_proc, end_return,* and *note_after_setjmp*, however, we maintain a model of the call stack. This model keeps track of the frame pointer and procedure id of each stack frame instance. Let p be a pointer and s be its source, that is, the stack memory location containing p. We find the model for the stack frame instance containing p by searching between which two frame pointers s falls. This gives us the procedure id and the offset, which is all we need to access the information in the stack frame type model from the run-time analysis.

4.4 Using accuracy for globals

Like for the stack accuracy, we make use of the instrumentation that gets inserted anyway. At program startup (see figure 2.1), we call *note_global* to record the mapping from id numbers to global object addresses. From the information of the first run, we also have a mapping from id numbers to models for global objects. Normally, when BDW starts a garbage collection, it pushes (among others) the global data areas on the mark stack. If we use the accuracy for globals, we intercept this and instead push all targets of global pointers on the mark stack.

Pushing targets of global pointers works as follows. We iterate over all models for global variables. These have an id number and a bit vector of offsets containing pointers. From the id number, we get the pointer to the global object (instead of just its model), and can access the memory location holding a candidate pointer p. If we got so far, then we already know that $\neg$*use-accuracy(k)* $\vee$ *is-pointer*$_{\mathrm{Ana},k}(p)$ is **true**. Now we trigger BDW's own pointer/non-pointer distinction logic *is-pointer*$_{\mathrm{BDW}}(p)$ which will eventually lead to the pointed-to heap object being marked and pushed on the mark stack for further scanning.

4.5 Other modifications

Using accuracy accounts for a number of changes we made in the Boehm-Demers-Weiser garbage collector, as described above. However, there were some more things we modified.

4.5.1 Excluding static roots

As mentioned above, BDW starts off a garbage collection by pushing the roots (registers, stack of activation records, and global data areas) on the mark stack. The global data areas, or *static roots*, presented a difficulty. The instrumentation for either run introduces a number of globals which BDW should not consider as static roots. The garbage collector solves this for its own globals by putting them in one **struct** *GC_arrays*, so that they come consecutively in memory, and then explicitly excluding that area from the static roots. We copied this technique, and have a **struct** *gcinfoGlobals* of our own, likewise excluding it from static roots.

Surprisingly, BDW does not consistently put all its globals into *GC_arrays*. This seems inconsistent, and actually turned out to be a bug. In one of our test runs, we observed an otherwise dead heap object being incorrectly marked from a global variable of BDW intended for a different use. We reported this to Hans Boehm, who has since fixed it, and also fixed it in our own modified version of BDW.

4.5.2 Debugging output

The original garbage collector has a lot of places where it can print debugging information. Whether or not to generate this verbose output is steered by compiler flags and `#ifdef`s. This was inconvenient for actually debugging, so we converted the static `#ifdef`s to dynamic **if**s that check the debugging level requested at the command line.

4.5.3 Converting macros to functions

Some of the core functionality of BDW resides in function macros. Unfortunately, that is the very code we needed to step through with the **gdb** debugger. So we converted a number of function macros to normal C functions. In some cases, this involved adding a level of indirection for parameters where BDW relied on the call by name parameter passing semantics.

4.5.4 Collecting results

To collect the results presented in this thesis, we had to adapt some of BDW's own bookkeeping to our needs, and add some of our own. We will see details of this in section 6.2.

Chapter 5

Benchmarks for the Experiments

Table 5.1 lists the benchmarks used for the experiments. The two smallest programs, gctest and gctest3, are distributed along with Barlett's mostly copying garbage collector [4, 5]. Anagram, ks, ft, yacr-2, and bc are taken from Austin's pointer-intensive benchmark suite. Bshift is an Eiffel program compiled to C with SmallEiffel. Gzip and sed are common GNU utilities (bc is also a common GNU utility, but we took the source code from Austin's benchmark suite). Two of the largest programs, li and ijpeg, are integer benchmarks from the SPEC95 suite.

Table 5.1: Benchmarks, static characteristics.

Name	Source	Lines	Kind of Program
gctest3	Barlett [3]	85	synthetic test
gctest	Barlett [3]	196	synthetic test
anagram	Austin [2]	647	string processing
ks	Austin [2]	782	graph algorithm
ft	Austin [2]	2 156	graph algorithm
yacr-2	Austin [2]	3 979	logic design
bshift	Hirzel	4 398	object-oriented
bc	Austin [2]	7 308	calculator/interpreter
li	Spec95 [19]	7 597	functional/interpreter
gzip	GNU [14]	8 163	compression
sed	GNU [14]	8 957	string processing
ijpeg	Spec95 [19]	31 211	image compression

5.1 Description of the Benchmarks

The description of the Austin benchmarks is based heavily on the README file in their distribution [2].

5.1.1 gctest3

This constructs a large number of heap objects and tests that arrays of pointers, interior pointers, and lists remain consistent. There is no explicit deallocation. For our experiments, we adapted it to use the Boehm-Demers-Weiser garbage collector instead of Barlett's garbage collector. We also reduced the number of objects created to make our runtime analysis feasible.

5.1.2 gctest

This constructs a lot of lists, trees, and 1 000 Byte arrays on the heap and tests their consistency after garbage collection. Like for gctest3, we changed the garbage collector it uses and reduced the problem sizes.

5.1.3 anagram

An anagram is a word or phrase formed by reordering the letters of another word or phrase. This benchmark generates anagrams. It does a lot of character pointer arithmetic and is very recursive. It uses the library routine *qsort()*, which gets passed a pointer to a non-library function. This confused the model of the call stack that we maintain in the runtime analysis, so it had to be replaced by our own implementation of quicksort.

5.1.4 ks

The Kernighan-Schweikert graph partitioning tool does a lot of pointer and array dereferencing and arithmetic, some dynamic storage allocation, and no explicit dynamic

storage deallocation. The only changes to the source code were to let it use our garbage collector.

5.1.5 ft

Ft finds minimum spanning trees. It does a lot of dynamic allocation and explicit deallocation, but very little pointer arithmetic. The only changes to the source code were to let it use our garbage collector. We are using a larger workload than Austin

5.1.6 yacr-2

This benchmark, yet another channel router, is used for integrated circuit layout. It does a lot of pointer and array dereferencing and arithmetic, some dynamic storage allocation, and no explicit dynamic storage deallocation. The only changes to the source code were to let it use our garbage collector.

5.1.7 bshift

This Eiffel program computes characteristics of the barrel shifter regular network topology. It makes use of inheritance and virtual functions. We compiled it to C using SmallEiffel and replaced the garbage collector that usually comes with the runtime system by our garbage collector.

5.1.8 bc

This is the GNU bc calculator. It implements a reference counting scheme internally for number and abstract syntax tree nodes. We did not remove this; however, we can select to disable explicit deallocation in our framework. Bc has some functions with variable argument lists, which we rewrote to take a fixed number of arguments. We are using a larger workload for this than Austin.

5.1.9 li

This is an xlisp interpreter. It dispatches calls through a global array of function pointers. Li has its own garbage collector, which we replaced by our modified Boehm-Demers-Weiser collector. Also, we rewrote variable argument functions to take a fixed number of arguments.

5.1.10 gzip

A commonly used compression program. It does not allocate much dynamic memory, but it deals with all kinds of values that might look like pointers, which makes it interesting in our context. We used it for decompression, since for compression it did no calls to *malloc()* at all.

5.1.11 sed

The stream editor is a batch program that transforms texts via regular expressions. To get it to work with our infrastructure, we had to rewrite variable argument functions. Also, we compiled it with the option to not use *alloca()*, a simple garbage collector scheme, but to call *malloc()* instead which gets handled by BDW.

5.1.12 ijpeg

This program does image compression/decompression on in-memory images based on the JPEG facilities. Like bc, it has some functions with variable argument lists, which we rewrote to take a fixed number of arguments. Ijpeg does explicit allocation and deallocation; we hooked it up with our garbage collector and can, like for the other programs, choose to ignore the explicit *free()*.

5.2 Memory usage

Table 5.2 describes the benchmarks with their actual workloads. Allocation is the number of bytes allocated over the whole run of the program (the total size of all objects if we never deallocated anything). The heap size is the largest number of bytes we saw at any point during program execution for the least effective deallocation scheme (out of the different garbage collector configurations and the explicit deallocation).

Table 5.2: Benchmarks, dynamic characteristics.

Name	Workload	Allocation	Heap size
gctest3	loop to 20,000	3 600 008	2 030 024
gctest	only repeat 5 in listtest2	1 749 416	360 800
anagram	`words < input.in`	265 984	265 704
ks	`KL-2.in`	15 840	15 840
ft	`8000 16000`	2 391 000	2 382 376
yacr-2	`input2.in`	267 512	249 688
bshift	scales 2 through 12	1 189 976	117 520
bc	find primes smaller 1000	24 260 608	762 984
li	`boyer.lsp`	7 669 920	292 600
gzip	`-d texinfo.tex.gz`	28 376	15 608
sed	strip unistd.h	60 056	24 592
ijpeg	`penguin.ppm`	172 448 632	10 159 504

Chapter 6

Experimental Results

6.1 Overview

For each benchmark, we count how many bytes are live in objects at any point in time. Time here is the total number of bytes allocated by the benchmark so far. Therefore, the line graphs all have "saw-teeth": as data gets allocated, live-size and time increase at the same pace. As data gets deallocated, live-size drops without time proceeding as you cannot allocate and deallocate at the same time. Note that we count the number of bytes in live objects, which is different from (less than) the number of bytes in live heap blocks or even the whole memory footprint. This must be kept in mind when viewing the plots, since for example the BDW garbage collector with its segregated storage manages a lot of "unused" heap space, but also the explicit allocator will not be able to avoid fragmentation entirely. For a discussion of this point, see [27].

In the graphs in this chapter, we use the following abbreviations:

gc	conservative Boehm-Demers-Weiser
s	accuracy for pointers from the stack
h	accuracy for pointers from the heap
g	accuracy for pointers from global objects
free	explicit deallocation

These are used in an additive notation. For example, gc+shg stands for the garbage collector with all accuracy information, but with *free()* disabled; gc+free stands for the

conservative collector with *free()* enabled; and free stands for only explicit deallocation without any garbage collector.

We present the results with two kinds of graphs. The ones showing live-size which look like flatirons are explained above. The other kind of graphs are scatter plots. They show the ratios of live sizes for different configurations as compared to the conservative BDW directly after garbage collections. A ratio less than one means that the configuration in question was more effective than BDW. For example, figure 6.3 (b) shows that both free and gc+g were about 7% more effective than gc at one of the garbage collections.

Out of our twelve benchmarks, for eight of them there is no difference between explicit deallocation, conservative garbage collection, and accurate garbage collection. For gctest3 and gctest, the Barlett benchmarks, this is not surprising, since they contain no explicit *free()* and use only pointers or small integers. Anagram and ks, the smallest benchmark from Austin, just build up data structures that never become unreachable. The benchmarks bshift and li are an Eiffel program compiled to C and a lisp interpreter, respectively. They are designed for use with a garbage collector and thus have no calls to *free()*. Still, it was not obvious that accuracy would make no difference.

6.2 Measurement details

Internally, at every call to *malloc(), calloc(), realloc(),* and *free()*, we keep track of the total number of bytes allocated so far and the total number of bytes freed so far. Since we are adding a header word to heap allocated objects, we subtract these 8 Bytes out of the collected statistics to get the correct numbers. After each call to *free()*, we write the time and the number of freed bytes out to disk.

After each garbage collection mark phase, we iterate over all heap blocks managed by the Boehm-Demers-Weiser collector. For each block b, we look up the number of live objects n_b and the size of the objects in this block s_i. Even though the sweep phase has

not yet taken place, n_b already reflects the state after the complete collection. Then we calculate

$$livesize = \sum_{b \in heap\text{-}blocks} n_b \cdot (s_i - 8)$$

The -8 correctly accounts for the header word we add to each heap object. We write the time, the garbage collection number, and the *live-size* out to disk.

Note that we are using the size of the objects as seen by BDW, which might be rounded up from the size of the objects as originally requested by the user program (see page 3.2).

For ease of comparison, we disallow garbage collections to be triggered implicitly. Instead, we garbage collect after fixed time intervals r. Let t be the number of bytes allocated so far. We trigger a garbage collection before doing *malloc(s)* if and only if $t \not\equiv t + s \bmod r$. This enables us to draw fair conclusions on the ratio of live-sizes for different configurations at the same point of time.

Since those configurations that disallow explicit deallocation have fewer data points than those where statistics get written out after each *free()*, we reconstruct the live-size for the missing points of time. This is trivial due to the linear increase given time as number of bytes allocated. We also reconstruct the number of bytes live immediately before a deallocation, since we only record the number after that deallocation. Finally, to get the numbers for explicit deallocation without garbage collection, we do a run with garbage collection and just subtract the total number of bytes freed so far from the total number of bytes allocated so far.

6.3 Ft and bc

Although these benchmarks display an interesting heap allocation behavior, there is no difference in the effectiveness of the various configurations at garbage collection times. We let the garbage collections be triggered implicitly instead of forcing it at fixed

intervals, and use the results to shed some light on the unmodified behavior of BDW and on our measurements.

Figure 6.1 shows the results for ft. We see that there is a garbage collection after 1 691 504 bytes of allocation which reclaims only a tiny amount of storage, and another one at program exit. We explicitly inserted the garbage collection at program exit. It shows that BDW is able to reclaim all the storage that the programmer returned with *free()*, but it does not do so as timely as the programmer did. What we do not see is that there were also garbage collections after 205 776, 413 824, and 672 976 Bytes of allocation: those did not find any dead objects.

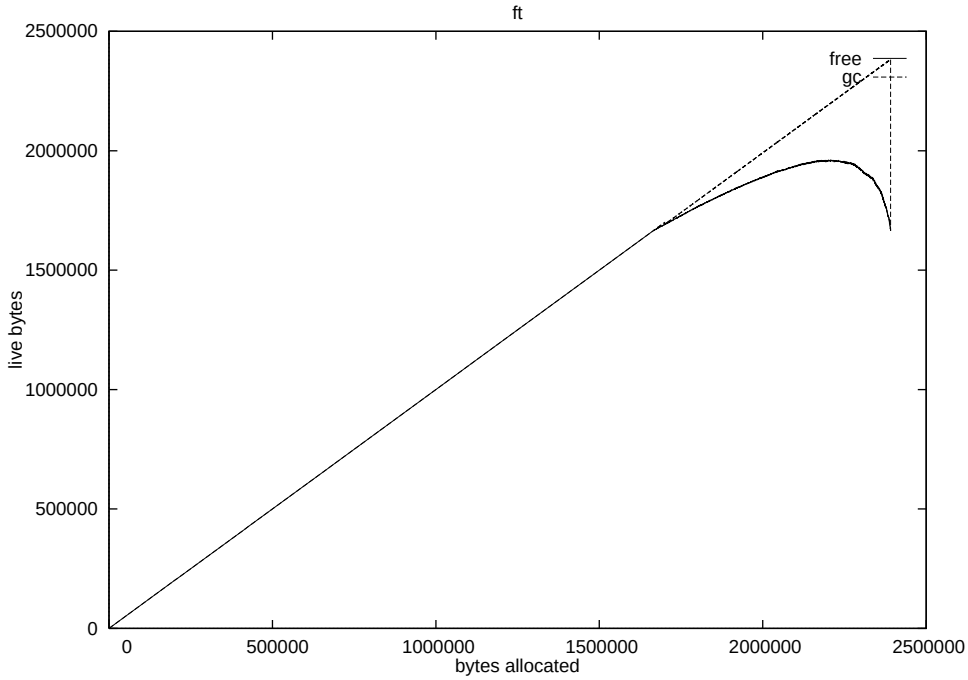


Figure 6.1: Results for ft.

Figure 6.2 shows the results for bc. Note that the slope of the individual lines looks higher only because the x-axis has a much smaller scale than the y-axis; in reality, all slopes in all our graphs are either $+45\%$ or -90%. Having this in mind, we can also read the seemingly fat line for free for what it is: a lot of tiny saw teeth, one per call to *free()*. This graph shows that BDW gets called quite frequently, since the heap keeps

growing to about double the live-size if you disable explicit deallocation. Periodically, BDW increases the heap size, which allows it to collect less frequently for a while. Note also that the garbage collection at the end of the program, which, again, would not have happened without our explicitly inserting it, reclaims more memory than the programmer explicitly frees.

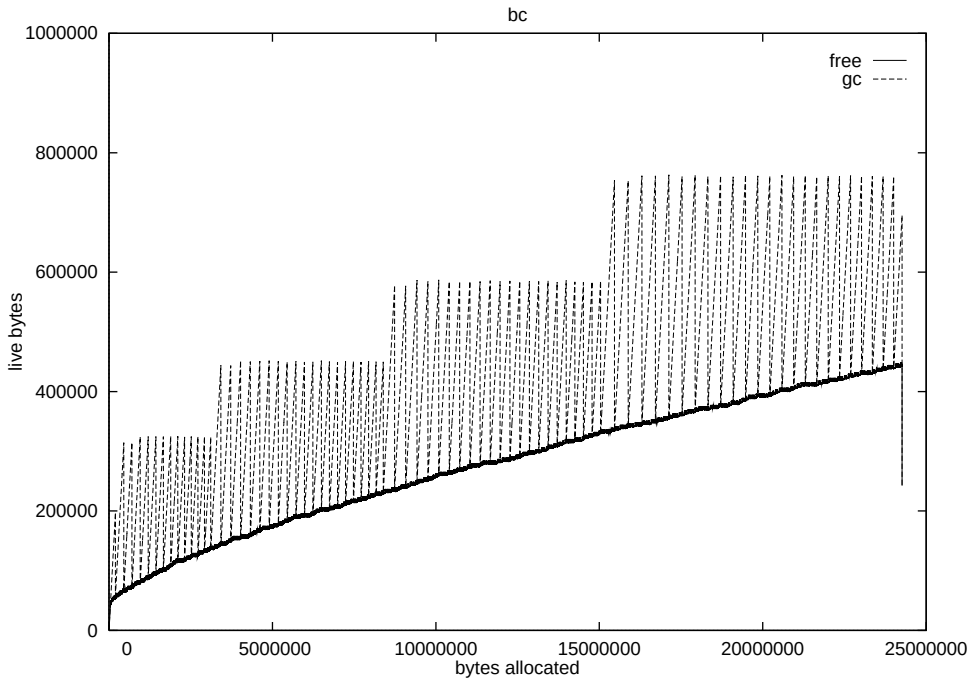


Figure 6.2: Results for bc.

These graphs demonstrate that even if you abstract from fragmentation issues and even if accuracy makes no difference, the use of a garbage collector may lead to a much larger memory footprint. That is caused by the fact that garbage collections tend to reclaim memory later than explicit deallocation. In the rest of this chapter, we ignore these timeliness issues and concentrate on the live-size directly after garbage collections, instead.

6.4 yacr-2

For this benchmark, explicit deallocation is more effective than accurate gc, which in turn is more effective than conservative gc. Figure 6.3 shows the results.

The first and only *free()*s happen after 228 544 Bytes of allocation and gain 38 632 Bytes. For the garbage collections, we disabled explicit deallocation. The collections after 8 648, 128 736, and 199 704 Bytes do not reclaim anything. After 249 688 Bytes, the garbage collector with accurate information reclaims almost as much as the explicit *free()* did, while the conservative collector does less well. At the very end of the program, the garbage collector finds objects to reclaim which the programmer did not bother to return.

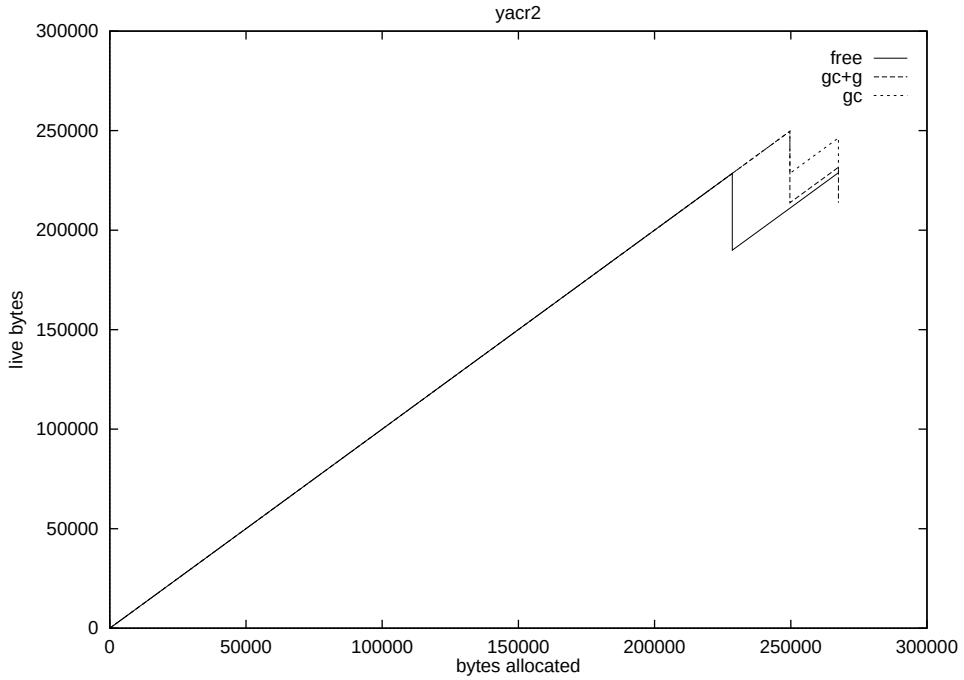
We found that the only accurate information that mattered here was that for global objects. That is, with accurate information about heap or stack only, the effectiveness of the conservative collector was not improved. Where the conservative collector was least effective, 92.35% of the data was left alive by *free()*, and 93.40% was left alive by the accurate collector. That means that for this benchmark, while there were differences, they were not excessive.

6.5 gzip

This program exhibits the strongest differences in effectiveness for the various configurations. While explicit deallocation repeatedly frees all live heap objects, accurate collection never quite reaches that, and conservative collection performs even worse. Figure 6.4 shows the results.

At all times, there is only a small number of live objects; even with the conservative collector, there are never more than 46. Allocation happens at such a coarse granularity that explicit deallocation falls neatly together with the garbage collections. Since we explicitly collect garbage before allocations that increase the number of allo-

(a)



(b)

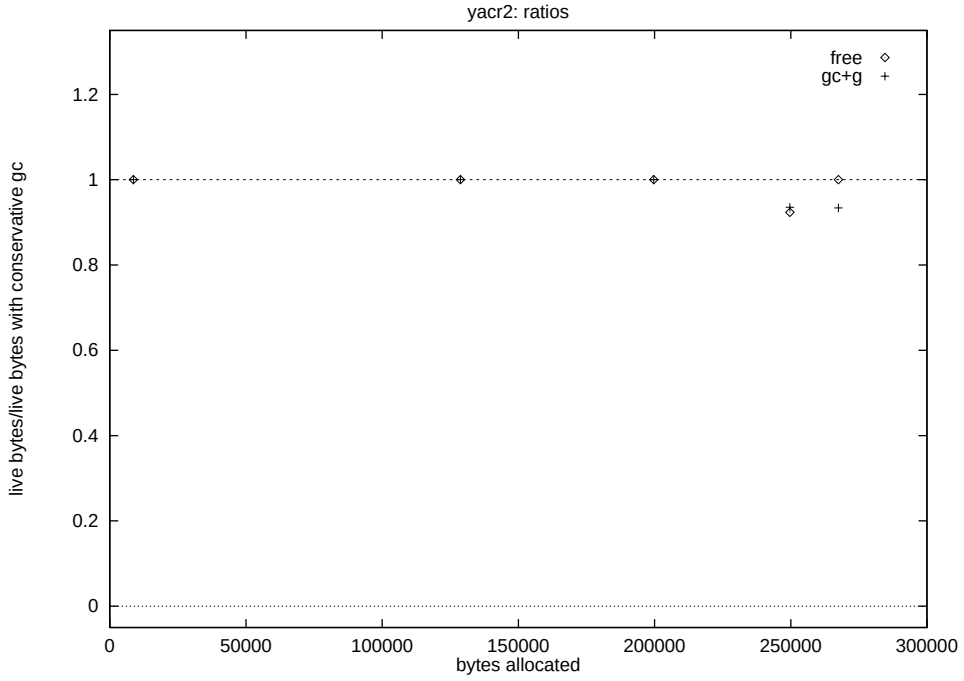
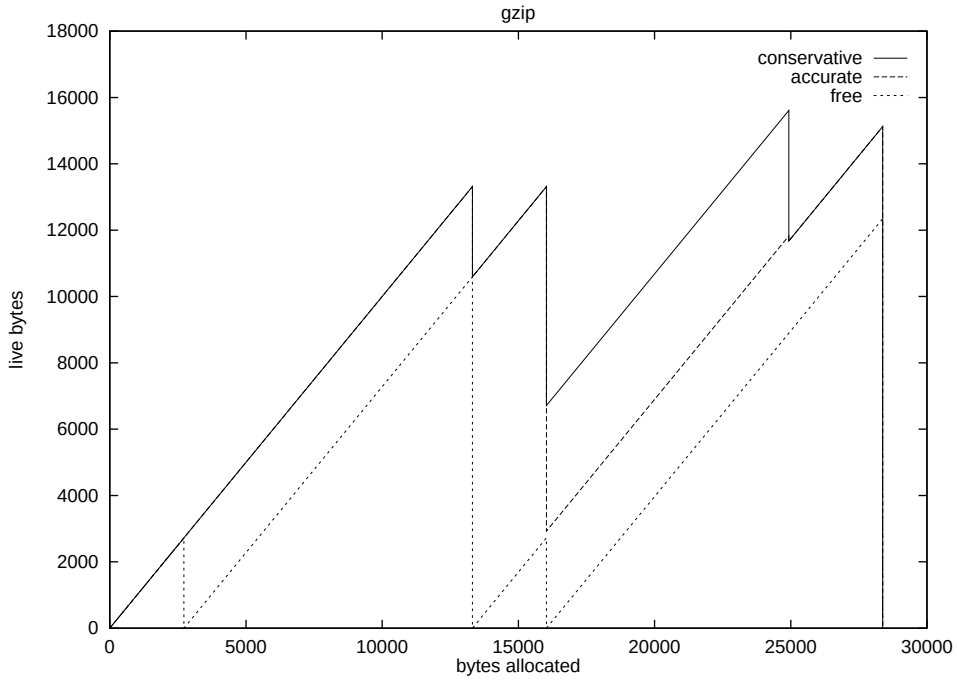


Figure 6.3: Results for yacr-2.

(a)



(b)

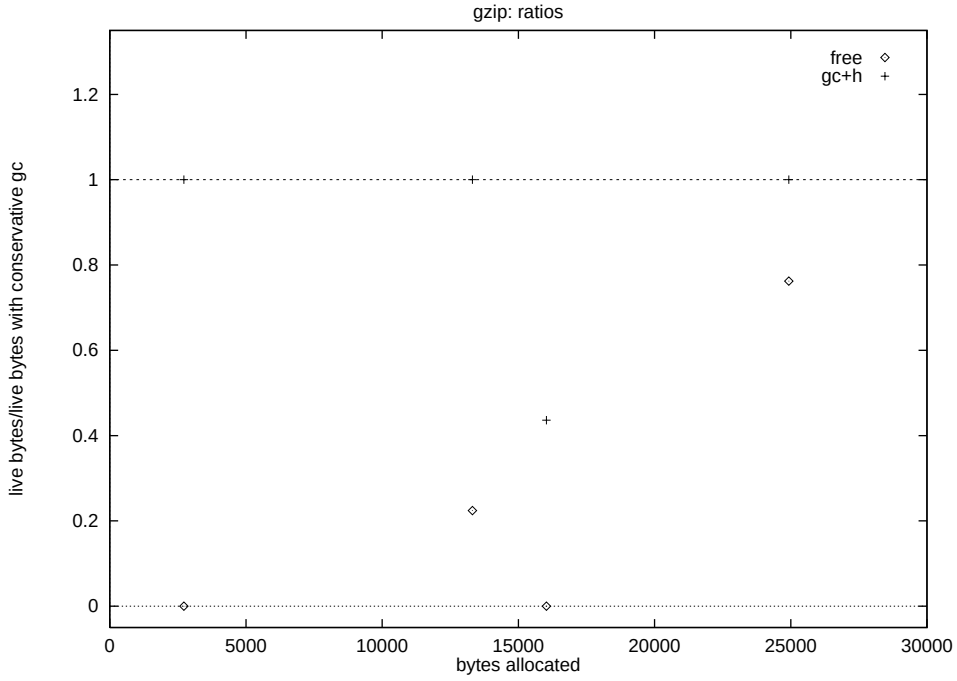


Figure 6.4: Results for gzip.

cated bytes over a threshold, we only get a regular gc "rhythm" if the size of individual objects is small compared to the total allocation, which is not the case here. The accuracy matters only one time, for the collection after 16 032 allocated Bytes. Here, the conservative collector keeps 46 objects around, while the accurate collector only keeps 3 objects with 2 928 Bytes total.

The scatter plot in figure 6.4 (b) shows how large a fraction of Bytes *free()* and accurate collection keep alive as compared to the conservative collector. This time, the only area for which accurate information mattered was the heap. We see that the accurate collector retains 43.62% of the data the conservative collector does at the one point where there is a difference.

6.6 sed

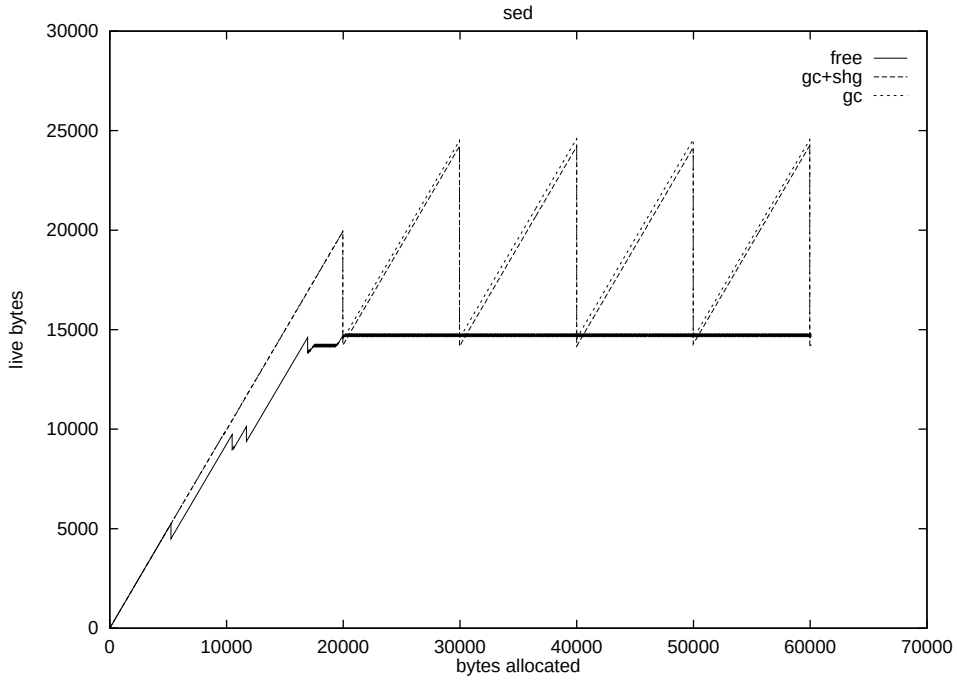
The original heap memory behavior of sed is that it builds up some data structures until it reaches a plateau, and then calls *malloc()* and *free()* at about the same rate. Figure 6.5 shows the results.

Again, accurate collection is more effective than conservative collection. What is new is that most of the time, both garbage collection mechanisms are even more effective than explicit deallocation. This means that there is a small memory leak in the sed benchmark.

At the second garbage collection after 9 808 bytes of allocation, accurate garbage collection is marginally better than conservative collection, with a ratio of 99.84%. At the same time, explicit deallocation is visibly better than conservative garbage collection, with a ratio of 92.56%. We also ran the program with accuracy and *free()* both enabled, and found that they supplemented each other, yielding a ratio of 92.40%.

The results for the garbage collections 3-7 is quite similar. With only explicit deallocation, the number of live bytes is consistently ca. 100.93% of that for conservative garbage collection at gc points. With accurate garbage collection, the number of live

(a)



(b)

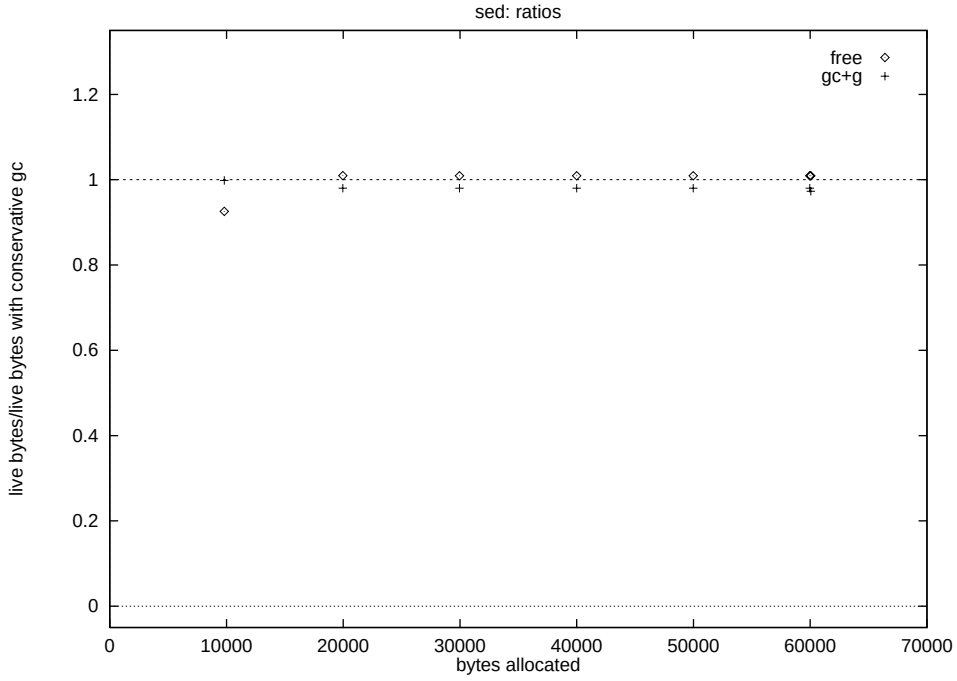


Figure 6.5: Results for sed.

bytes is consistently ca. 98.02% of that for conservative collection. We see that the differences are only marginal for this benchmark.

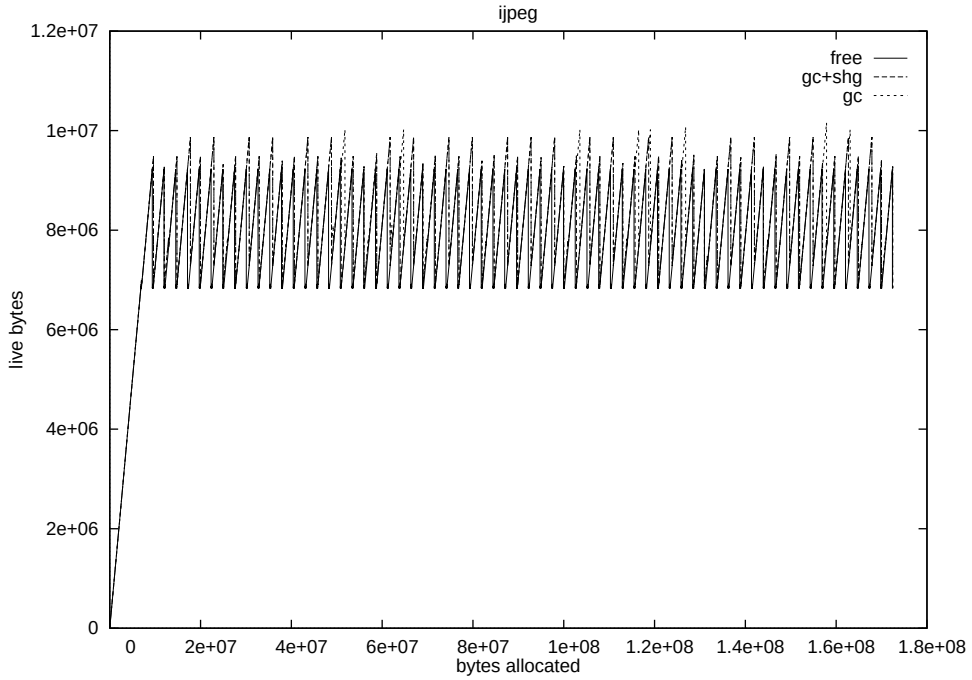
6.7 ijpeg

This benchmark allocates with 164 Megabyte the largest amount of heap storage, which unfortunately makes our figure 6.6 hard to read. Surprisingly, the behavior of the original program without garbage collector is very similar to that with garbage collector.

Here, it happens occasionally that *free()* perform worse than either accurate or conservative garbage collection. Otherwise, accurate garbage collection and explicit deallocation are identical.

Most of the time, accurate gc is either as bad as or only slightly better than conservative gc in terms of effectiveness, meaning that it keeps 99-100% of the data alive that conservative gc does. But there are exceptions where it keeps only 74.47% as many bytes around, meaning that the conservative collector encountered values it wrongly interpreted as pointers to a lot of live data. Those values all resided in global objects.

(a)



(b)

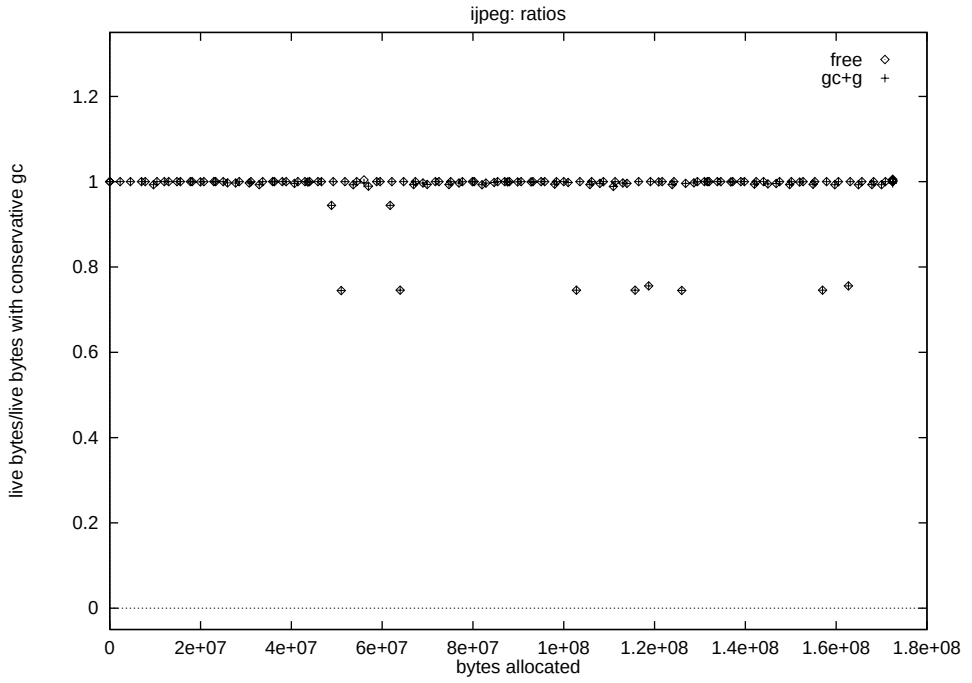


Figure 6.6: Results for ijpeg.

Chapter 7

Related Work

7.1 Barlett 1988

The technical report that describes Barlett's mostly copying garbage collector [4] includes an interesting evaluation of its effectiveness. It compares four variants of the garbage collector for Scheme that differ in their accuracy for stack roots. All variants assume perfectly accurate information for the heap. Barlett's GC-0 is stack accurate, the only pointers it is not sure about reside in registers. Barlett's GC-2 is about as conservative for the stack as the Boehm-Demers-Weiser collector we use, it considers each aligned stack word that seems to point into some object a pointer.

For both his benchmarks, Barlett finds that GC-0 is more effective than GC-2, but only slightly so. He records the fractions of the heap retained at each garbage collection, but they cannot be directly compared for the different garbage collector variants since the garbage collections are not triggered at the same time. Another difficulty is that it is not clear what exactly is the metric for heap size.

This paper complements our findings. In our experiments, we never saw an effect of stack accuracy, and we never saw differences in gc effectiveness for programs compiled from or interpreting other languages than C. GC-0 corresponds to our gc+shg and GC-2 to our gc+hg configuration, and thus Barlett's results give samples for a difference between these.

7.2 Zorn 1993

Zorn's technical report [27] compares BDW to four widely used explicit allocators. Zorn measures the CPU time, memory usage, page fault rate and cache miss rate. For the performance metrics, he finds that BDW performs about as well as the other algorithms.

In principle, Zorn uses the configurations gc and free that we considered in our experiments. He does not directly evaluate effectiveness, though. Instead of counting bytes in live objects, he gives the whole memory usage of the allocator. This focuses on exactly those fragmentation issues of the specific allocators which we abstracted from. He finds that under this metric, BDW uses 30% to 150% more storage than explicit deallocation. One curious note is that Zorn reports yacr to have a serious memory leak. For our benchmark yacr-2, this has obviously been fixed, since we find that BDW does not reclaim more storage than *free()*.

This paper nicely complements our experiments. We did not consider the CPU performance, since such measurements are meaningless on our instrumented programs. Zorn did not consider effectiveness or accuracy, since he focuses on the total memory footprint of his benchmarks.

7.3 Other garbage collector comparisons

Hicks, Moore and Nettles [15] compare different copying garbage collectors for efficiency. They find that low-level optimizations can gain a remarkable speedup. The scope of their paper includes garbage collection on heaps of Java and SML programs on different architectures. On the other hand, they are not concerned with effectiveness: the only accuracy configuration they use is total accuracy, abbreviated as gc+shg in our results chapter.

Smith and Morrisett's paper [18] introduces a mostly-copying garbage collector

MCC which reuses and adds to Barlett's ideas. MCC and BDW are similar in their conservativism, MCC even allows inaccurate information for the heap in some objects. Smith and Morrisett report that their garbage collector's overall performance is better than that of BDW, but that it also results in a larger overall memory footprint.

7.4 Allocation and deallocation behavior

Zorn and Grunwald [28] study the *malloc()*s and *free()*s in six C programs, all but one of which Zorn used one year later in the technical report [27] discussed above. They find that there are usually many small and few large objects, many short-lived and few long-lived objects, and many objects of only a few object sizes. Surprisingly, they even find that there are significant clusters of inter-arrival times of allocations, where time is measured in cycles.

Stefanović and Moss [21] investigate the object behavior for SML programs. Like Zorn and Grunwald, they also find that there are many short-lived and few long-lived objects, but they come to that conclusion with a very different methodology. Stefanović and Moss count the total volume of bytes that survive at least one garbage collection, and vary the parameter that determines how frequently garbage collections need to happen. That means that lifetime of objects is measured in bytes of allocation, not cycles, and an object becomes dead as soon as it is unreachable. The lowest object lifetime they can measure is 32 000 Bytes, and more than 98% of all objects die before they reach that age.

7.5 Adding accuracy

In [12], Diwan, Moss and Hudson describe the technique of *stack maps*. They explain how even for type-safe languages like Modula-3, due to optimizations finding the accurate root pointers may be difficult. As a solution, the compiler generates maps that, for each point at which garbage collection is allowed to happen, help recover

accurate pointer/non-pointer for the stack.

The stack map technique has also been used by Agesen, Detlefs, and Moss [1], and by Stichnoth, Lueh, and Cuerniak [22] for supporting garbage collection in Java Virtual Machines. Both papers describe that one can easily incorporate a limited form of liveness information into the stack map. Consider a local variable p that will never be dereferenced in the future. Let o be the object pointed to by p. If no other pointers point to o, then o can be safely reclaimed even though it is still reachable.

Furthermore, these last two papers also describe solutions to the *jsr* problem. The problem is that for one special situation in Java byte-code, you cannot statically determine which stack slots hold pointers and which don't. In other words, you do not have stack accuracy, which you need for copying garbage collection. They solve this by *splitting* variables or by dynamically finding the missing information.

Chapter 8

Conclusion

8.1 Summary

Table 8.1 summarizes the results from our experiments. For each benchmark and accuracy configuration, we look at the quotient of the live-size with that configuration and the live-size with the conservative BDW. The table reports the minimum and maximum of this quotient across all garbage collections that happened during the benchmark run. We sorted the benchmarks differently in this table than in tables 5.1 and 5.2 for clarity of exposition.

Table 8.1: Summary of Results.

	$\frac{\text{gc+s}}{\text{gc}}$		$\frac{\text{gc+h}}{\text{gc}}$		$\frac{\text{gc+g}}{\text{gc}}$		$\frac{\text{free}}{\text{gc}}$		$\frac{\text{gc+shg+free}}{\text{gc}}$	
Benchmark	min	max	min	max	min	max	min	max	min	max
gctest3	1	1	1	1	1	1	designed for gc		designed for gc	
gctest	1	1	1	1	1	1	designed for gc		designed for gc	
bshift	1	1	1	1	1	1	designed for gc		designed for gc	
li	1	1	1	1	1	1	designed for gc		designed for gc	
anagram	1	1	1	1	1	1	1	1	1	1
ks	1	1	1	1	1	1	1	1	1	1
ft	1	1	1	1	1	1	1	1	1	1
bc	1	1	1	1	1	1	1	1	1	1
gzip	1	1	0.4362	1	1	1	0	1/0	0	1
yacr-2	1	1	1	1	0.9340	1	0.9235	1	0.9235	1
sed	1	1	1	1	0.9731	1	0.9256	1.0094	0.9240	1
ijpeg	1	1	1	1	0.7447	1	0.7447	1.0059	0.7447	1

We see that stack accuracy never mattered for our benchmarks. Note, however,

that we did observe stack accuracy effects for the example program in figure 1.1).

The most extreme results are those for gzip, where heap accuracy yields a 56.38% gain over the conservative collector. In that benchmark, we also see situations where explicit deallocation frees everything (min 0). At the end of the program, conservative collection frees everything (max 1/0); this last collection is an artifact of our experiments and would not happen in reality, though.

Surprisingly, most often, accuracy for globals is what improves the garbage collector, by up to 25.53% for ijpeg. So far, we do not have an explanation for this.

Doing both garbage collection and explicit deallocation has an effect. Sometimes, at one garbage collection point, gc is better than free, and at the next point this relation is reversed. Using the combined approach gives us the better behavior at all times.

The bug mentioned on page 4.5.1 was found by observing inconsistencies in reported statistics. We found it because we were wondering how free could perform better than gc+free.

8.2 Reflection

The numbers show that for the benchmarks we used, it does make a huge difference whether you have a conservative garbage collector, an accurate garbage collector, or just do *free()* manually.

There still remain other considerations, though. When you compile or interpret programs from languages where the garbage collector comes with the language definition, not collecting garbage is not an option. When you want to do copying collection or support gc in the presence of orthogonal persistence, conservativism is not an option (you can still do mostly-copying collection, though). We conclude that gaining a little more accuracy for BDW is not an important goal, but coming up with complete accuracy, at least for some memory areas, would be worthwhile.

Let us review where our runtime analysis did not find the true "limit" (see section

2.4). It propagates pointerness through operands like $-, <, \leq$ etc., but these result in small numbers which get filtered by BDW's pointer/non-pointer distinction. It is flow insensitive, but that corresponds to a type analysis and is quite realistic. It is context-insensitive, meaning that we expect stack frame instances of the same type to have pointers at the same offsets. This last point might be important, especially since we did not see stack accuracy effects on our benchmarks for our concept of accuracy.

8.3 Future work

There are a number of ideas that are logical continuation of this thesis.

- Modify our runtime analysis so that it corresponds to a context-sensitive pointer analysis. Then see whether this suggests stronger effectiveness variations.
- Modify our runtime analysis so that it also does a limit study for pointer liveness. How much more effective could the garbage collector be if it knew exactly which pointers will be dereferenced in the future?
- Do the same liveness-limit-study for Java.
- Investigate ways to provide accuracy for C. Starting points are the stack map, splitting, and dynamic type recording and recovery techniques reported in the literature for Modula-3 and Java [1, 12, 22].

Bibliography

[1] Ole Agesen, David Detlefs, and Eliot Moss. Garbage collection and local variable type-precision and liveness in java virtual machines. In PLDI'98, pages 269–279, May 1998.

[2] Todd Austin. Pointer-intensive benchmark suite. http://www.cs.wisc.edu/~austin/ptr-dist.html.

[3] Joel Barlett. Garbage collector sources. ftp://gatekeeper.dec.com/pub/DEC/CCgc/.

[4] Joel Barlett. Compacting garbage collection with ambiguous roots. Technical report, Western Research Laboratory, Digital Equipment Corporation, February 1988.

[5] Joel Barlett. Mostly-copying garbage collection picks up generations and C++, technical note TN-12. Technical report, Western Research Laboratory, Digital Equipment Corporation, October 1989.

[6] Jeffrey Barth. Shifting garbage collection overhead to compile time. CACM, pages 513–518, July 1977.

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

[8] Hans-J. Boehm, Alan Demers, and Scott Shenker. Mostly parallel garbage collection. In PLDI'91, pages 157–164, November 1991.

[9] Hans-J. Boehm and Mark Weiser. Garbage collection in an uncooperative environment. In PLDI'88, pages 807–820, September 1988.

[10] David Chase, Mark Wegman, and Kenneth Zadeck. Analysis of pointers and structures. In PLDI'90, pages 296–310, June 1990.

[11] Ramkrishna Chatterjee, Barbara Ryder, and William Landi. Relevant context inference. In POPL'99, pages 133–146, January 1999.

[12] 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.

[13] Maryam Emami, Rakesh Ghiya, and Lauri Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In PLDI'94, pages 242–256, June 1994.

[14] GNU. Gnu ftp list. http://www.gnu.org/order/ftp.html.

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

[16] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In PLDI'99, pages 105–118, January 1999.

[17] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In POPL'97, pages 1–14, January 1997.

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

[19] Standard Performance Evaluation Corporation SPEC. Cint95 benchmarks. http://www.spec.org/osg/cpu95/CINT95/.

[20] Bjarne Steensgaard. Points-to analysis in almost linear time. In POPL'96, pages 32–41, 1996.

[21] Darko Stefanović and Eliot Moss. Characterization of object behaviour in Standard ML of New Jersey. In LISP and Functional Programming, pages 43–54, June 1994.

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

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

[24] Paul Wilson. Uniprocessor garbage collection techniques. In International Workshop on Memory Management, pages 1–42, September 1992.

[25] Paul Wilson, Mark Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In International Workshop on Memory Management, pages 1–78, September 1995.

[26] Robert Wilson and Monica Lam. Efficient context-sensitive pointer analysis for C programs. In PLDI'95, pages 1–12, June 1995.

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

[28] Benjamin Zorn and Dirk Grunwald. Empirical measurement of six allocation-intensive C programs, CU-CS-604-92. Technical report, University of Colorado at Boulder, July 1992.