Compiler Support to Customize the Mark and Sweep Algorithm

Dominique COLNET, Philippe COUCAUD, Olivier ZENDRA
E-mail: {colnet, coucaud, zendra}@loria.fr

LORIA
UMR 7503
(INRIA - CNRS - University Henri Poincaré)
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex
FRANCE

Abstract

Mark and sweep garbage collectors (GC) are classical but still very efficient automatic memory management systems. Although challenged by other kinds of systems, such as copying collectors, mark and sweep collectors remain among the best in terms of performance.

This paper describes our implementation of an efficient mark and sweep garbage collector tailored to each program. Compiler support provides the type information required to statically and automatically generate this customized garbage collector. The segregation of objects by type allows the production of a more efficient GC code. This technique, implemented in SmallEiffel, our compiler for the object-oriented language Eiffel, is applicable to other languages and other garbage collection algorithms, be they distributed or not.

We present the results obtained on programs featuring a variety of programming styles and compare our results to other high quality garbage collectors.

1 Introduction

In the last few decades, automatic memory management gradually and constantly improved, and now seems to be preferred to manual memory management in most modern programming languages. Numerous and efficient techniques [Wil92, WSNB95, JL96] have been developed, providing a wide range of solutions to language implementors.

Nonetheless, some developers still consider that the best performance can be reached only by relying on manual memory management. Indeed, they believe it enables them to better address the specific memory requirements of their application. However, the garbage collection community has been working on the customization of the GC to each application for a long time, thus addressing the concerns of proponents of manual memory management.

Many of these customized collectors [Bar90, Ede92, BS93, AFI95] still require some kind of intervention from the developer. In this paper, we present our experience with the implementation of a completely automatic system in which compiler support allows the generation of a customized GC without requiring any additional work from the application developer.

The GC we integrated to SmallEiffel — The GNU Eiffel compiler — is a partially conservative [Boe93] mark and sweep collector. We use an efficient type inference algorithm [CCZ97] to analyze the class relations at compile time. This algorithm provides the required information to segregate objects by type and statically customize most of the GC code. Thus, memory management relies heavily on type-accurate, efficient routines.

The remainder of this paper is organized as follows. Section 2 explains the overall method used to customize the GC code thanks to compiler support and then details the management of fixed-size objects. Resizable objects are considered in section 3. Section 4 briefly describes more specific, language-dependant optimizations. Performance measurements are presented in section 5. Finally, section 6 reviews related work and section 7 concludes.

2 Fixed-size object management

2.1 Allocation

The allocator we implemented takes advantage of object structure information, provided by the type inference process [CCZ97] of SmallEiffel. Because it statically knows which kinds of objects are allocated, fixed-size objects are segregated by type, rather than by size as in most other segregated algorithms [WSNB95].

A specific collection of typed chunks is dedicated to each inferred live type. This way, a chunk holds only one type of objects (see figure 1), whose size is known at compilation time and hard-coded wherever it is needed. Thus, the SmallEiffel GC does not need any extra word to store the object size. Each chunk is a memory area which contains a fixed number of slots for objects — not references

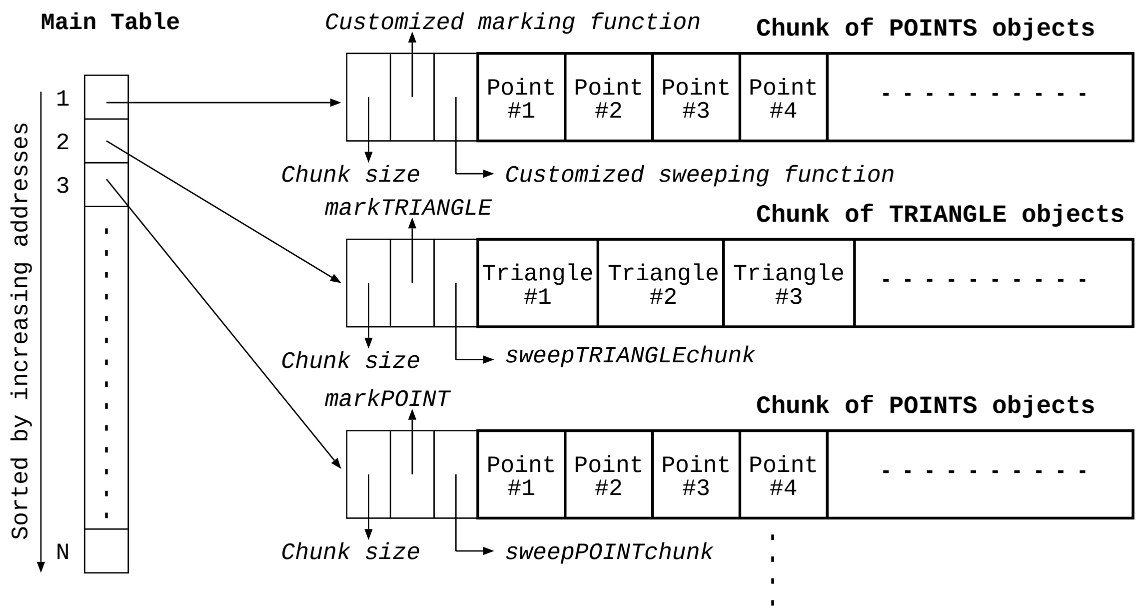

Figure 1: Overview of Memory Layout

— of the corresponding concrete type. A chunk header comprises some other type-specific information like pointers giving access to customized marking and sweeping functions.

After trying and benchmarking some configurations, we found that fixed-size chunks of 8Kb were a good tradeoff between fast allocation and tight memory footprint.

Each type has its own customized allocation method and is associated to a Linear Allocation Chunk (LAC). The latter is a chunk managed in a stack-like way, with a Free-Space Pointer (FSP) pointing to the beginning of the free space available for objects of this type. When an object must be created, its type allocation method tries to allocate it directly from the corresponding LAC, just by incrementing the FSP for this type by the object size. Such a linear segregated allocation is probably the fastest one can imagine.

When not enough memory is available in the LAC, the allocation method looks in the type free list, which chains free slots across all the chunks of the corresponding type (see figure 2). If the list is not empty, the first slot it refers to is removed from the list and used for the new object. The segregation of objects by type thus makes it possible to look for a free slot with a constant, low cost.

If no room can be found in any of the chunks corresponding to this type, a garbage collection cycle can be triggered, which should reclaim unused objects and thus provide a slot for the new object. In case the GC cycle does not provide the necessary memory, a new LAC for objects of the required type has to be malloc'd.

In order not to trigger a full GC cycle whenever no free room can be found either in the LAC or in the type list of free objects, an additional criterion, the memory "ceiling" is considered. It represents the headroom for fixed-size objects, that is the amount of allocated memory under which no garbage collection is requested, but a new chunk is malloc'd instead.

Thanks to the type inference performed at compile time, an initial value can be assigned to the ceiling by considering the number of live types in the system. For example, a program with a few concrete live types has a lower ceiling than another one with many live concrete types. In practice, the ceiling is equal to four times the number of live concrete types, which means each type is expected to use four chunks on average. Some Eiffel objects having specific properties in the system (uniqueness for example) are managed in a particular way (see section 4).

The constant ceiling incrementation is of course too simplistic to provide good performance, particularly because it does not consider the amount of memory previously allocated. Polynomial extrapolation seems well adapted because it is able to update the ceiling according to the previous evolution of memory requirements, even when a very steep increase occurs. However, we obtained the best results, both in terms of speed and memory footprint, with a simple, constant growing factor of thirty per cent. Thus, after each garbage collection cycle, the program is allowed to allocate new chunks representing up to thirty per cent of the amount of used chunks, in order to ensure it has enough headroom.

Figure 3 illustrates the behavior of this ceiling in a test program which features three different execution phases. In the first phase (GC cycles 0 to 9), it allocates a lot of memory. During this phase, the ceiling is quickly increased (it is doubled after each GC call when the allocated memory is less than 10Mb, and increased by thirty per cent after this limit). In the second phase (GC cycles 10 to 14), the program allocates objects without keeping references to them: the ceiling is not updated because the GC recycles enough memory chunks.

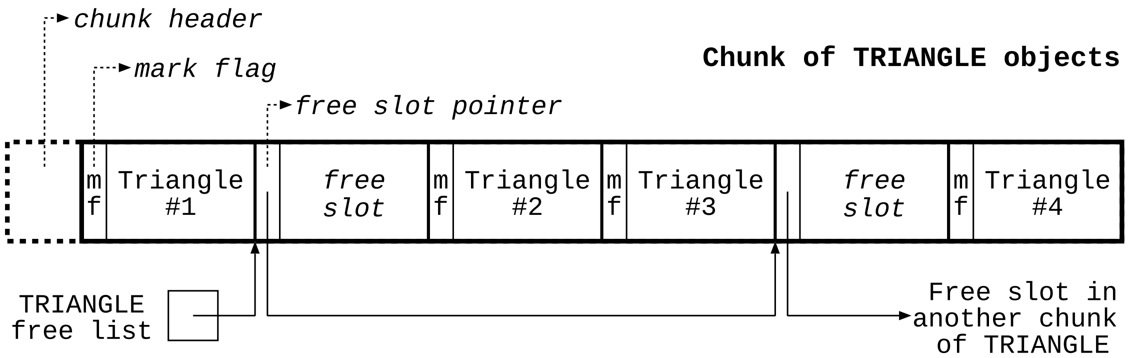

Figure 2: Detail of a fixed-size object chunk

Finally, the program enters a new allocation phase (GC cycles 15 to 18) which leads to a new series of 30% ceiling updates.

2.2 Marking

The marking process requires the addition of some information to each object in order to know whether it is marked or not. This extra information is implemented with a one-word long GC header for each object. It holds the mark-bit, which we implemented as an integer flag for the sake of simplicity. In future releases, we plan a more compact implementation using bitmap headers associated to each chunk.

The mark phase relies on two different steps depending on where object references are found: root pointers in the stack or internal pointers in object structures — in SmallEiffel C code, pointers may not be located in static areas.

2.2.1 Finding the roots

Of all the data contained in the stack, only references to objects are of interest for marking.

In Eiffel, normal objects are always allocated in the heap and, in case a normal object is referenced from a local variable or argument, only a pointer to its location is pushed into the stack. However, Eiffel's expanded objects [Mey94] may be allocated directly in the stack. They may hold non-reference objects, and also references.

Thus, examining the whole stack should allow all references to live heap objects to be retrieved. Finding internal references to other live objects will be explained later.

The stack depth[1] is an important factor to take into consideration. A good computation of the stack area to be analyzed may save some word analysis at each call to the GC. SmallEiffel considers the address of the object associated to the root class as the bottom of the stack, and the last local variable allocated as its top.

Since stack elements may be directly stored in processor internal registers, references to objects may remain outside the stack. Thus, beside the stack scanning process, another one is needed to access the processor registers. People interested in details may want to look at SmallEiffel source code, accessible from `http://www.loria.fr/SmallEiffel`.

When accessing any stack (or register) word, one does not know whether it is an Eiffel reference or another data type (a properly-aligned bit pattern). Consequently, all stack words are *a priori* considered possible references [BW88]. We thus need to efficiently identify genuine references: as in [Cha92] and [KID+90], four successive filters are used to reduce cases of misidentification.

Let r be the candidate reference, and N the total number of chunks (whatever the type). We note $[B_x, E_x]$, with $x \in [1, N]$ the range of addresses included in chunk x, and $ObjectSize(x)$ the size of the slots in chunk x.

1 - Because the addresses of all created chunks are sorted in the main table (as shown in figure 1), we immediately have the range of acceptable addresses for an object reference: r is an acceptable reference if and only if $r \in [B_1, E_N]$.

2 - Check if the potential pointer is included in the address range of an existing chunk:
$\exists i \in [1, N] \mid r \in [B_i, E_i]$.

3 - Check if the pointer actually refers to (the beginning of) an object in this chunk i. This is quickly done by a specialized function associated to the chunk, which verifies whether the pointer value corresponds to an offset from the beginning of the chunk by an integer multiple of the object size of the chunk:
$\exists k \in \mathbb{N}^+ \mid r = B_i + k \times ObjectSize(i)$.

4 - Check whether the pointed address corresponds to an unmarked allocated object, thanks to the object mark flag.

If the candidate reference passes these four tests, it is considered a valid root reference and the pointed object is marked live, using the GC header extra word.

It should be noted that after the above four tests we still do not know for sure whether a pointer from the stack really is a reference to an object or not. A coincidence might occur where a stack word contains a bit pattern which is a valid address, although the word is not a pointer at all. In this *misidentification* case, the object is maintained live when it should not be, and its mem-

[1] We consider here that addresses increase as the stack grows. But of course, SmallEiffel analyzes the direction of stack growth and handles both increasing and decreasing addresses.

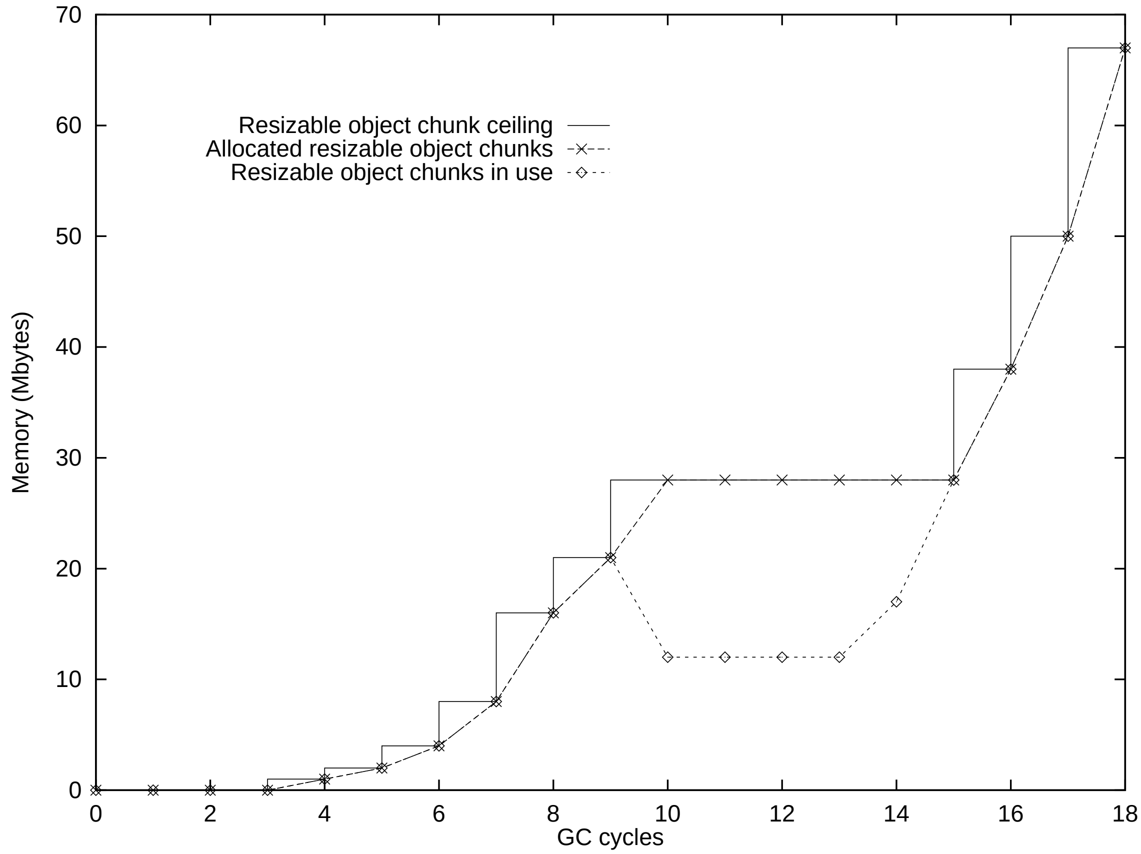

Figure 3: Updating the memory allocation ceiling.

ory block cannot be reused until the address disappears from the stack. This may cause a slight increase in memory use, but is the only safe policy, because it guarantees the completeness of the marking algorithm. In [Boe93], Boehm experiments on test programs showed that misidentifications caused memory retention of about 10% for fully conservative collectors. Memory retention is likely to be lower for our *partially-conservative* GC, because misidentifications can only occur when marking from the stack.

2.2.2 Following internal references

When an object is reached and marked live, the marking process must continue on all objects referred from it (its suppliers [Mey94]). A marking process that would know nothing of the object apart from its size would have to perform the same kind of reference identification as previously described for root detection. Of course, in probably all implementations of object-oriented languages, objects hold information about their dynamic type (if only to be able to perform late binding), in the form of a pointer to an object descriptor or as a type ID used to access a type descriptor table. The marking process is thus able to access the object description, and then follow each internal reference, without having to check whether the reference is a valid one or not. Many potential memory leakages [Boe93] are avoided in this way.

SmallEiffel, thanks to its capacities for type inference and code customization, implements this processing of internal pointers in a very efficient way.

Indeed, SmallEiffel — relying on important code customization — generates a specialized mark function for each object type. Such a function exactly knows where to find valid references to other objects and what their type is. So after the first blind jump from the stack into a chunk c, the customized marking function associated to chunk c (see figure 1) is called. Since it is a typed customized function, the marking process follows unambiguous, typed pointers and statically calls marking functions until a leaf (a childless object) is reached.

Assume for example that part of a system is composed of TRIANGLE objects which contain an INTEGER representing their color and three references to POINT objects, the latter holding two DOUBLEs as their coordinates. The marking function `markTRIANGLE` looks like this:

```
void markTRIANGLE(Triangle *triangle) {
    if (triangle->mark_flag!=MARKED_FLAG){
        triangle->mark_flag=MARKED_FLAG;
        if (triangle->point1 != NULL)
           markPOINT(triangle->point1);
        if (triangle->point2 != NULL)
           markPOINT(triangle->point2);
        if (triangle->point3 != NULL)
           markPOINT(triangle->point3);
    }
```

```
}
```

As can be seen, since attribute `color` of class TRIANGLE is known to be a non-reference field, no code needs to be generated for its marking in `markTRIANGLE`.

Since class POINT has only two DOUBLE attributes, and does not hold any reference attribute, no marking code needs to be generated for children:

```
void markPOINT(Point *point) {
    point->mark_flag=MARKED_FLAG;
}
```

Also note that the absence of children makes it unnecessary to test whether the flag is marked or not.

Of course, because of polymorphism, the concrete type of a given supplier may be any descendant of its static type, and a late binding on the correct marking method is required. SmallEiffel type inference mechanism [CCZ97] makes its possible to significantly reduce the cost of late binding, by narrowing the number of possible types to those which are actually live, ensuring a fast late binding. Assume for example that the abstract class FRUIT has two concrete living types: APPLE and PEACH. The implementation of late binding on marking functions for objects of static type FRUIT is as follows:

```
void markFRUIT(Fruit *fruit) {
    switch (fruit->id) {
       APPLEid: markAPPLE((Apple*)fruit); break;
       PEACHid: markPEACH((Peach*)fruit); break;
    }
}
```

As for any late binding function, such a method is in fact implemented using binary branching code, which results in very fast execution, as shown in [ZCC97].

2.2.3 Avoiding recursive marking

As usual in recursive marking algorithms, deeply nested structures such as very long linked lists tend to make the execution stack grow dramatically, eventually causing stack overflow.

The classic technique for solving this problem is to transform recursive calls into iterative loops and auxiliary data structures. Since the SmallEiffel compiler knows object structures, it can reorder the marking of their fields in the most efficient order, avoiding the use of any extra data structure when only one of these fields is the beginning of a long chain of references. For example, a linked list of INTEGERs is marked in this way:

```
void markLINK(Link *link) {
    do {
       if (link->mark_flag!=MARKED_FLAG){
         link->mark_flag=MARKED_FLAG;
         link = link->next;
       }
    } while (link != NULL);
}
```

2.3 Sweeping

The next phase, sweeping, consists in looking at all allocated objects in order to collect the memory used by those which have not been marked live.

We thus have to scan all the chunks of the main table (see figure 1) to collect the objects that are no longer referenced and whose flag has been left `unmarked`. This is efficiently done in SmallEiffel with sweeping functions customized for each type. Hence, the addresses where the flags are to be found can easily be computed, thanks to the fact that all objects in a chunk have the same predefined size. The continuous nature of the memory held by a chunk is also likely to guarantee a better data locality when scanning the chunk than with chained, scattered memory blocks. Here is a simplified example of a sweeping function customized for the TRIANGLE chunks:

```
void sweepTRIANGLEchunk(TriangleChunk *tc) {
    Triangle*tp;
    for (tp=tc->first; tp <= tc->end; tp++){
       if (tp->mark_flag!=MARKED_FLAG)
         addToFreeListOfTRIANGLE(tp);
       else
         tp->mark_flag=UNMARKED_FLAG;
    }
}
```

Each type has its own free list comprising all the free slots associated to this type, whatever chunk they are in. When an unmarked object is found, it is linked ahead of the type free list. This free list does not incur any space overhead, since the GC header extra word previously used to mark whether the object was live or not is now reused to chain the object to the free list.

All objects marked live are unmarked when the chunks are swept, readying the object graph for the next garbage collection.

When a chunk contains only free slots, it is put back in the list of free chunks. The latter is untyped, which allows a better recycling of chunks, since they all have the same size.

2.4 Finalization

Before an Eiffel object is collected, a finalization routine [Hay92] may be called on this object.

Finalization routines in the SmallEiffel GC are generated like all other user-defined routines. Thus, they are produced only for objects which actually define an effective finalization routine.

Since all these routines are known at compile time, the GC can be adapted to generate the corresponding calls when appropriate. Thus there is no need to check whether each object has to be finalized. Only the chunks holding objects which have to be finalized need to be examined, and in these chunks, it is easy to consider only the objects which have been marked `to_finalize`. In this way, the overhead incurred by finalization management is very limited and does not lower the program performance.

Although these ideas have not been implemented yet in SmallEiffel, future versions of our GC will feature such a customized finalization mechanism.

3 Resizable object management

Resizable objects are implemented in a very similar way to fixed-size objects. However, because the size of resizable container objects (arrays, strings) is not computable at compile time, these objects are more difficult to manage.

Small resizable objects are allocated like fixed-size objects (see section 2.1), in fixed-size typed chunks (see figure 4). The size of the latter, however, has been extended to 32Kb, in order to take into account the fact that resizable objects are generally larger than fixed-size ones. Resizable object allocation maintains its own ceiling, which is managed is the same way as that of fixed-size objects, except that the ceiling initial value is four times the number of statically computed live concrete types. Very large resizable objects which are larger than the normal size of these chunks are handled differently. Each of them constitutes a single "one-big-slot chunk", whose size is simply the resizable object size.

The number of elements contained by the resizable object is kept in its header. This makes it possible to mark the container contents, if they are references to objects. This also allows the sweeping of resizable object chunks, since the position of each of the resizable objects in the chunk can easily be computed.

The marking mechanism is very similar to that of fixed-size objects. However, for each element of a resizable container, there is no extra data at all. Instead, a single common extra word is associated to the container. Thus, in arrays and the like, a memory slot fits exactly the contained object to which it is allocated.

As for fixed-size objects (see figure 1), there is one specific, customized marking function for each live concrete type corresponding to a resizable container. Thus, since ARRAY[INTEGER] and ARRAY[TRIANGLE] are two distinct concrete types, they require different marking functions.

When the container holds reference objects, the generated marking function must propagate the marking process to each element. For example, the marking function for a container of TRIANGLEs consists of a loop which propagates the marking process to the elements:

```
void markContainerOfTRIANGLE(Triangles *c) {
    int i;
    if (c->mark_flag!=MARKED_FLAG){
      c->mark_flag=MARKED_FLAG;
      for (i=c->count - 1; i >= 0; i--) {
         Triangle *t = c->storage[i];
         if (t != NULL) markTRIANGLE(t);
      }
    }
}
```

Since this function has been customized it is very efficient. Indeed, the most appropriate code for the element type is used instead of a generic one: direct calls to `markTRIANGLE` are made while marking a container of TRIANGLEs.

For a container holding non-reference objects, such as an array of integers, no further marking is needed. Hence, the marking function just marks the container itself and returns:

```
void markContainerOfINTEGER(Integers *c) {
    c->mark_flag=MARKED_FLAG;
}
```

On such a container, the SmallEiffel customized marking algorithm is faster than that of a conservative garbage collector.

Sweeping of resizable objects is similar to that of fixed-size objects, except that the former must take into account the actual size of each container object in the chunk. Since this process does not rely on the element type of the container, the same sweeping function is used for all resizable object chunks.

When a resizable object is collected, it is put in the free list corresponding to its type. As for fixed-size objects, a completely free chunk is put in an untyped free list of chunks and may be reused for any type.

When a very large resizable object is freed, the corresponding chunk is also put in this free list. This chunk may then be split to be used as a normal-size chunk for small resizable objects. A consequence of this splitting is that the remainder may constitute a smaller than normal chunk. To avoid excessive fragmentation, a coalescing is periodically triggered on these chunks.

4 Specific optimizations

The peculiarities of some kinds of objects result in various specific customizations.

4.1 The root object

The *root object*, the first object created, on which the root method of the system is called, lives as long as the program. Thus, it cannot be garbaged and always remains marked. However, since its attributes may change during the program execution, marking them is still necessary.

Furthermore, since the root object is frequently the only object of its type in the system, it is not allocated in a normal chunk, but apart, to avoid wasting memory.

4.2 *Once* function results

Once routines [Mey94] are a specificity of Eiffel. The body of such a routine is executed only once in the program lifetime, the first time the routine is called. Subsequent calls return without executing the routine body.

When the *once* routine is a function, its result is computed the first time the routine is called, and returned at each subsequent call.

Hence, objects returned by *once* functions, or *once objects*, live from the time they were allocated till the end of the program. Thus, exactly like the root object, a once object must always be considered as marked by the collector, and may not be collected.

Once object management can thus be optimized by relying on specific marking and completely avoiding sweeping. This is currently being implemented in SmallEiffel.

For some simple *once* functions, it is possible to unambiguously know the type of the result at

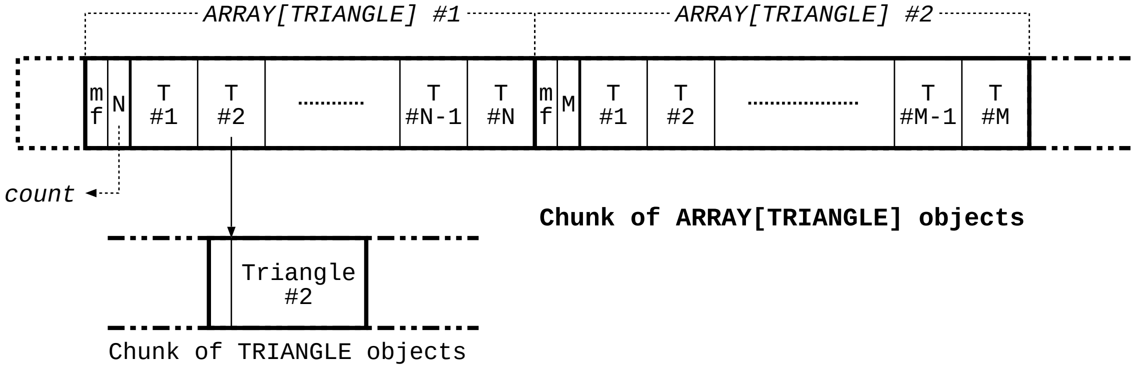

Figure 4: Detail of a resizable object chunk

compile time. Such results can thus be precomputed, that is created at the very beginning of the program [ZCC97]. Further optimizations, such as not checking whether the object is NULL, can be performed in the GC when dealing with these objects.

4.3 Manifest strings

A *manifest string* is a string whose value appears directly in the source code. A manifest string is not a constant string; it is a reference to a resizable container of characters. Indeed, in Eiffel, the developer does not normally have direct access to a resizable object itself, but to a fixed-size object which encapsulates the behavior of the resizable object in a portable way and hides its implementation from the user [Mey94].

A manifest string can thus be considered as a special type of *once* function whose value is precomputable at compile time. Consequently, all the manifest strings are allocated in specific memory areas, are not subject to sweeping, and have a marking process customized and optimized as described for precomputable once functions. This is currently implemented in SmallEiffel.

5 Performance

In order to evaluate the performance of our GC, we chose to benchmark several implementations of an Othello (or Reversi) game. These programs had been designed by 24 teams of students using the previous version of SmallEiffel, without a GC. They enable us to compare real programs performing the same kind of task, with various *programming styles* and algorithms, resulting in different execution behaviors, especially with respect to memory. Other results are available in [CZC98], where we benchmarked small programs featuring various *execution patterns*, as well as the SmallEiffel compiler itself.

Two of the 24 Othello programs were incorrect (failing because of assertion violations) and thus could not be used. The 22 remaining programs can be split in two categories: non-leaky programs, where memory has been sparingly managed, and leaky ones, with many short-lived objects.

Being generated in ANSI C, our GC is platform-independent and has been tested on a wide range of UNIX, Macintosh and Windows platforms. For the sake of brevity, we present here the results we obtained with the aforementioned programs on one UNIX platform.

5.1 Benchmarking platform

We compared the customized mark-sweep GC generated by SmallEiffel to the Boehm-Demers-Weiser GC (BDW)[BW88]. The BDW GC is a renowned conservative GC which has been a topic of numerous research papers, e.g [Boe93, Zor93, DDZ94]. It has also been developed for a long time on UNIX systems and is thus very mature. Implemented in real-life systems, it is a fast, robust and slim GC, thanks to highly optimized algorithms and efficient data structures. The BDW GC is thus a very valuable reference system.

Furthermore, it has been used successfully for some time in conjunction with SmallEiffel, because the latter did not provide its own GC until version `-0.81`[2].

Since version `-0.80`, SmallEiffel is able to produce the C code adapted to an application with or without generating the C code corresponding to the customized GC, depending on whether option `-no_gc` has been selected or not. When the GC is also generated, instantiation instructions rely on the whole GC described in sections 2 to 4. Conversely, with `-no_gc`, the allocation routine of a new object (e.g `newTRIANGLE`) calls the standard C library `malloc`. This makes it easy to include an external GC library redefining `malloc` and `free`, like the BDW GC, or to use no GC at all.

We report here the results we obtained on a SUN Sparc Ultra Enterprise with 512 Mb of RAM, running Solaris 5.6. The large amount of memory of this machine made it possible for all the benchmarks — even the most memory-hungry — to run entirely in RAM without being swapped to disk.

SmallEiffel `-0.80` was used with optimization options `-boost` and `-no_split` to generate the C code. Of course, `-no_gc` was added when producing the C code to be linked with the BDW 4.12 library (for which option

[2] The first version of SmallEiffel was numbered `-0.99`; version `-0.80` is the 20th.

-Dall_interior_pointers was disabled). The C code was compiled with egcs-1.0.1 (a variant of gcc), using optimization option -O6.

Because of the differences of complexity between the various game algorithms used by the different teams, and because of the major differences in the efficiency of the implementation of these algorithms, we had to benchmark them on different board sizes in order to get the most meaningful results. Of course, each program was benchmarked with the same board size without any GC, with BDW and with SmallEiffel.

5.2 Non-leaky programs

Fourteen of the Othello programs were non-leaky ones, featuring rather careful object instantiations and reuse of objects whenever possible. This is consistent with the fact they were designed at a time when SmallEiffel did not provide its own GC mechanism. Figure 5 shows the results obtained on these programs.

5.2.1 Memory footprint

As could be expected, for all these non-leaky programs, the maximum memory footprint was roughly the same without GC and with either the BDW or the SmallEiffel GC. One of the 14 Othello produced a memory footprint around 2 Mb, all the others using about 1 Mb only.

In all cases, running the Othello program without any GC lead to the smallest memory footprint. Indeed, with non-leaky programs, a GC can collect very few — if any — garbage objects, whereas it does take space, since it requires extra code and data structures.

On 6 of the Othello programs, SmallEiffel incurs a smaller memory footprint than BDW, by 1 to 7%, whereas BDW has an advantage of 1 to 9 % on 7 programs. Both generally need about 25% more memory than the -no_gc version, which represents roughly 300 Kb on our benchmark programs. The performances achieved by the BDW and SmallEiffel garbage collectors are thus very similar, which indicates the validity of our approach.

5.2.2 Execution time

On these non-leaky programs, execution times without any GC or with either GC are generally alike.

Overall, the -no_gc version is the fastest, since it outperforms both GC versions in 7 out of 14 cases, with an advantage of up to 8% over the faster of the two GC versions. The BDW GC version is the quickest in 2 cases, whereas SmallEiffel scores first in 3 cases.

These results confirm that, for non-leaky programs, it is better not to use a GC at all, and that using either SmallEiffel or BDW generally results in a speed decrease, although a limited one.

When only the BDW and SmallEiffel are considered, the former is faster in 8 cases, by up to 20% (program #9). On the other hand, SmallEiffel offers a moderate speed advantage (up to 5%) in 4 cases. This clearly shows that the overhead incurred by the GC on non-leaky programs is lower in BDW than in SmallEiffel. This can be explained by tha fact that the SmallEiffel GC is still in its early days, and offers room for optimization.

5.3 Leaky programs

We also benchmarked 8 different leaky Othello programs, among which sloppy memory allocations and deallocations cause very important memory leaks — 37 to 194 Mb — in 6 cases. We think these programs constitute a benchmark which is more representative of typical memory usage when the developer relies on an automatic GC. Figure 6 shows the results obtained.

5.3.1 Memory footprint

The usefulness of a GC clearly appears on these leaky programs. Although the -no_gc versions take between 2 and 194 Mb of memory, all programs using either the BDW or the SmallEiffel GC feature a very reasonable memory footprint (between 1 and 2 Mb), similar to that of non-leaky programs. This demonstrates the effectiveness of both collectors.

When only BDW and SmallEiffel are considered, their performance in terms of memory footprint are alike. However, BDW generally has an advantage of 136 to 552 Kb, which on these programs featuring small optimal memory footprints translates to 10 to 31%.

5.3.2 Execution time

Overall (5 cases out of 8), both the BDW and SmallEiffel GC are faster than the -no_gc version. This is because less memory has to be allocated thanks to the recycling of dead objects by the system. The only case where the -no_gc version is the fastest — program #19 — also confirms the previous statement. Indeed, this program is by far the one where memory leakage is the smallest, less than 1 Mb, which thus limits the gain a GC can reap. On most programs, this gain is about 10% for both GC versions, and even reaches about 30% on the "leakiest" benchmark (team #16, with a 193 Mb leak).

The SmallEiffel GC performs well when compared to the BDW GC. The latter outperforms SmallEiffel in 3 cases, with an execution time advantage of up to 17% (on program #22), whereas SmallEiffel is faster in 5 cases, by up to 11% (program #16).

Consequently, although it does not have the maturity of the BDW GC, the SmallEiffel GC appears quite efficient and compares very well with this much renowned efficient automatic memory management system.

6 Related work

The research carried out by Detlefs *and al.* [DDZ94] tends to prove that conservative garbage collector performances compare quite well with explicit memory deallocation.

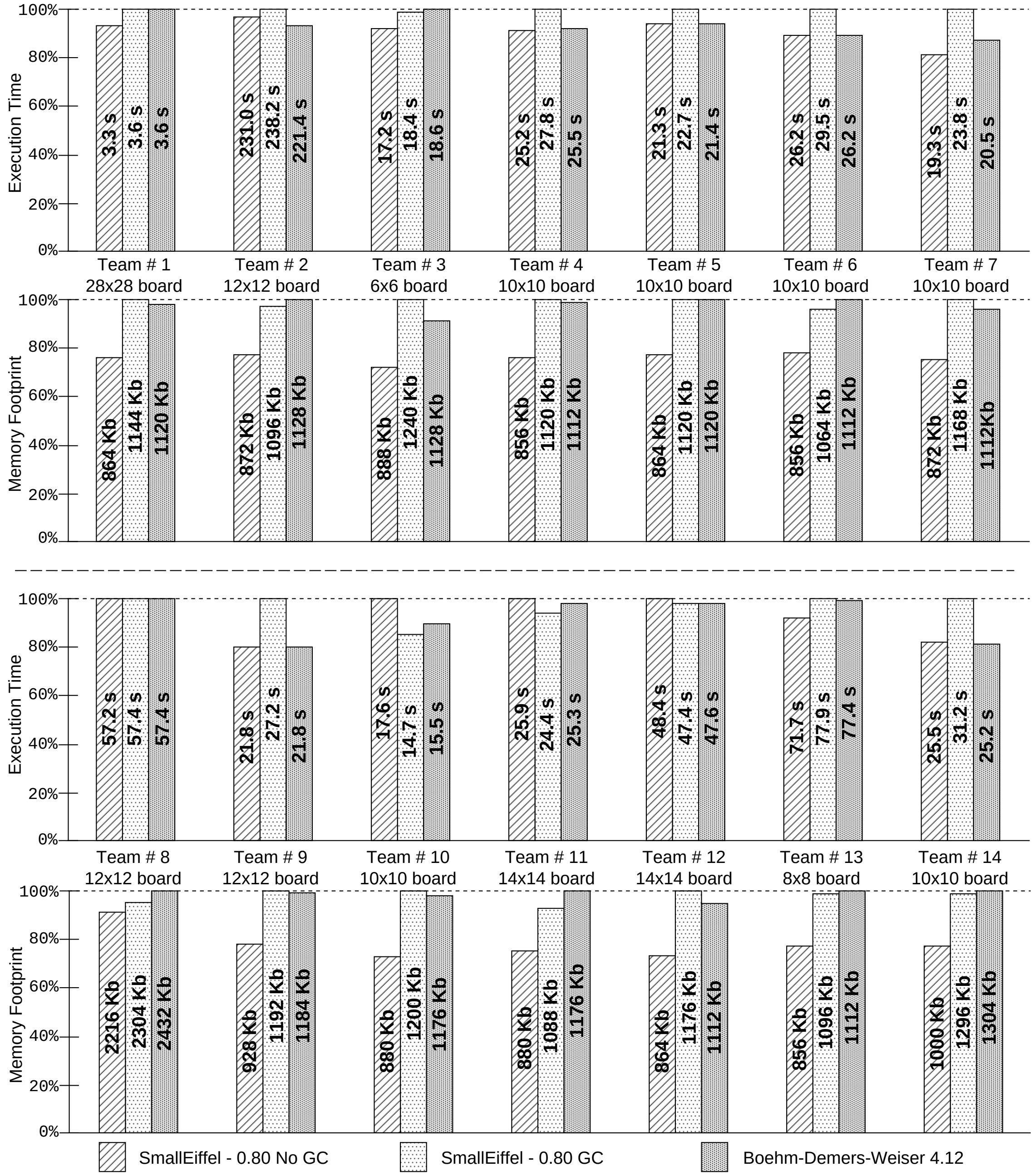

Figure 5: Execution time and memory footprint comparisons of non-leaky programs under UNIX

But although classic conservative collectors perform well, a little extra information about specific memory patterns is likely to significantly improve the results. This information may be provided by different sources: profiling, developer, type analysis, ...

Grunwald *and al.* describe in [GZ93] their CustoMalloc system. After the profiling of a program, CustoMalloc produces a customized memory allocator (malloc) and a customized memory deallocator (free) which are able to handle more efficiently the most frequent object sizes. Their work indeed shows on a variety of commonly used programs that a few classes of object sizes — generally small sizes — represent almost all the allocated memory. Unlike CustoMalloc, SmallEiffel generates not only customized memory allocator and deallocator, but a complete, customized GC system. Furthermore, since in our system only static analysis is used to provide the information needed for customization, no pre-run is required. It seems nonetheless possible that SmallEiffel might benefit from using profiling information, especially to predict resizable object sizes and the most common classes of fixed-size objects.

In [BS93], Boehm and Shao studied the performance of an enhanced version of BDWGC involving type inference during the collection phases,

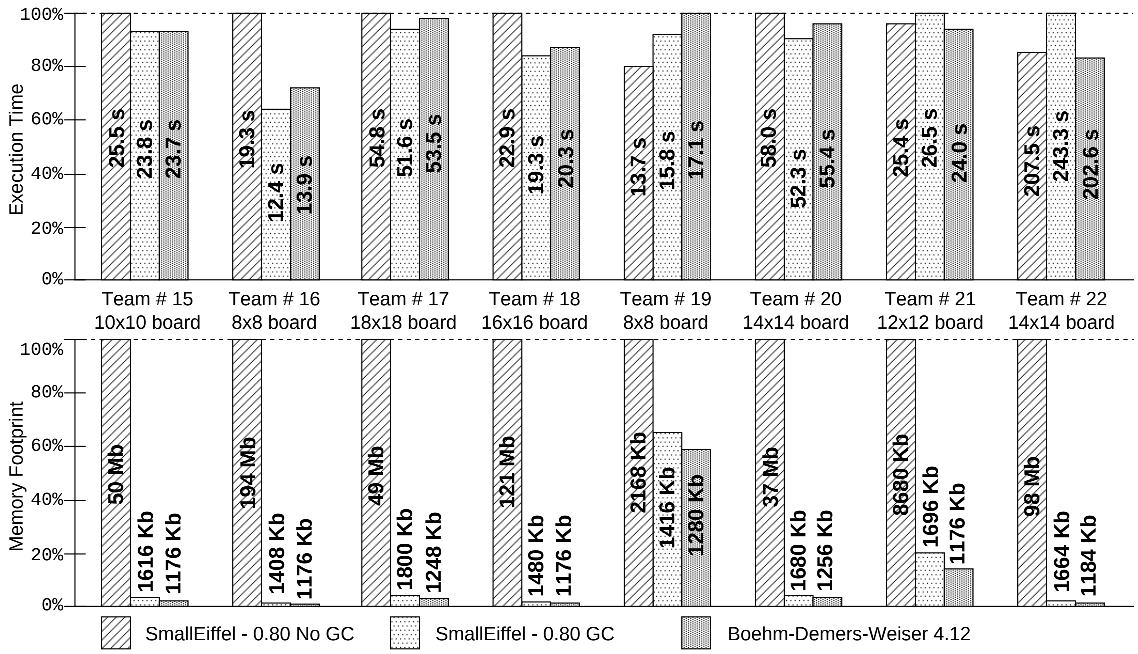

Figure 6: Execution time and memory footprint comparisons of leaky programs under UNIX

thanks to a user-typed malloc. At runtime, they sample the first several objects allocated for each type and infer a type map. This extra information allows a more accurate marking of the memory and offers better performance.

Bartlett's Mostly Copying collector [Bar88] is a hybrid conservative and copying collector. It assumes no knowledge of register or stack area layouts, but it does assume that all pointers in heap-allocated data can be found accurately thanks to the registration of all internal roots [Bar90] by the developer.

Other experiments around garbage collector customization were carried out in [AF94] and [AFI95]. Their Customizable Memory Management (CMM) allows users to customize object management by specifying at each object allocation which policy to adopt for its storage, and by providing the optimal traversal routines for each type. The major interest of the latter kind of manual customization is that it allows *type-accurate* memory management.

In Edelson's garbage collector for C++ [Ede92, Ede93], the marking functions are automatically produced by a preprocessor which generates a call to a marking function for each pointer in the class. This syntactical substitution reduces internal pointer misidentifications and speeds up the marking process.

Diwan *and al.* [DMH92], as well as Agesen and Detlefs [AD97], describe a kind of compiler-supported garbage collection which allows *accurate* (or *exact*) collection. To be able to find pointers in the stack and in registers at run-time, their compilers statically generate tables which encode the location of these pointers at any point where a collection might occur. At garbage collection time, the return addresses of stack frames give access to these tables.

An important difference between SmallEiffel and all previously described research is that SmallEiffel automatically generates typed, customized memory management functions. Thanks to the static type inference performed by SmallEiffel, no additional information, provided either by the developer or by pre-executions, is required.

7 Conclusion

We have presented an implementation of a compiler-supported customized mark-sweep garbage collector for an object-oriented language. Unlike many previous works, this customization is completely and automatically performed by the compiler, without any intervention from the developer.

The set of benchmarks we described in section 5, featuring various *programming styles*, allowed us to evaluate the performance of the GC on different memory patterns. The results obtained on these various execution patterns clearly show the validity of the approach both in terms of memory footprint and execution time.

In this paper, we described a compiler-supported GC customization technique for a mark and sweep algorithm in an Eiffel compiler. However, most of this technique is not specific to our system and is likely to be applicable to other class-based languages and other garbage collection algorithms, even distributed ones.

Although the SmallEiffel GC performs quite well, it could benefit from the addition a some features. Incrementality, for example, may be an important asset in some situations and is thus worth exploring.

Our future work is likely to focus on improving the performance of our GC. The addition of flow analysis or profile-guided analysis to SmallEiffel would provide the GC with more information on memory requirements, such as the most frequent *size classes*, thus helping to better tune the GC, and would allow an increased degree of customization of the GC routines. Deferred sweeping and coalescing of memory chunks is also likely to increase the GC performance, by delaying operations which are not immediately necessary.

Acknowledgments

We would like like to thank the anonymous reviewers for their helpful comments and suggestions. We are also grateful to Jean-Michel Drouet who proofread early versions of this paper.

References

[AD97] Ole Agesen and David Detlefs. Finding References in Java Stacks. In *OOPSLA'97 Workshop on Garbage Collection and Memory Management*, 1997.

[AF94] Giuseppe Attardi and Tito Flagella. Customising Object Allocation. In *ECOOP'94*, volume 821, 1994.

[AFI95] Giuseppe Attardi, Tito Flagella, and Pietro Iglio. Performance-Tuning in a Customizable Collector. In *International Workshop on Memory Management (IWMM'95)*, volume 986 of *Lecture Notes in Computer Sciences*, pages 179–198, 1995.

[Bar88] Joel F. Bartlett. Compacting Garbage Collection with Ambiguous Roots. Technical report, DEC Western Research Laboratory. 88/2, 1988.

[Bar90] Joel F. Bartlett. A generationnal, compacting collector for C++. In *ECOOP/OOPSLA'90 Workshop on Garbage Collection in Object-Oriented Systems*, 1990.

[Boe93] Hans-Jurgen Boehm. Space-efficient conservative garbage collection. In *PLDI'93*, volume 28 of *SIGPLAN Notices*, pages 197–206, 1993.

[BS93] Hans-Juergen Boehm and Zhong Shao. Inferring type maps during garbage collection. In *ECOOP/OOPSLA'93 Workshop on Garbage Collection in Object-Oriented Systems*, 1993.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[CCZ97] Suzanne Collin, Dominique Colnet, and Olivier Zendra. Type Inference for Late Binding. The SmallEiffel Compiler. In *Joint Modular Languages Conference*, volume 1204 of *Lecture Notes in Computer Sciences*, pages 67–81. Springer-Verlag, 1997.

[Cha92] Emmanuel Chailloux. Conservative Garbage Collector with Ambiguous Roots, for Static Type Checking Languages. In *International Workshop on Memory Management (IWMM'92)*, volume 637 of *Lecture Notes in Computer Sciences*, pages 218–229, 1992.

[CZC98] Dominique Colnet, Olivier Zendra, and Philippe Coucaud. Using Type Inference to Customize the Garbage Collector in an Object-Oriented Language. The SmallEiffel Compiler. In *OOPSLA'98*, 1998. Submitted.

[DDZ94] David Detlefs, Al Dosser, and Benjamin Zorn. Memory Allocation Costs in large C++ Programs. *Software Practice and Experience*, 24(6), 1994.

[DMH92] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler Support for Garbage Collection in a Statically Typed Language. In *PLDI'92*, volume 27 of *SIGPLAN Notices*, pages 273–282, 1992.

[Ede92] Daniel R. Edelson. Precompiling C++ for Garbage Collection. In *International Workshop on Memory Management (IWMM'92)*, volume 637 of *Lecture Notes in Computer Sciences*, pages 299–314. Springer-Verlag, 1992.

[Ede93] Daniel R. Edelson. Type Specific Storage Management. Technical report, Baskin Center for Computer Engineering & Informations Sciences, University of California, Santa Cruz, UCSC-CRL-93-26, 1993.

[GZ93] Dirk Grunwald and Benjamin Zorn. CustoMalloc: Efficient Synthesized Memory Allocators. *Software Practice and Experience*, 23(8):851–869, August 1993.

[Hay92] Barry Hayes. Finalization in the Collector Interface. In *International Workshop on Memory Management (IWMM'92)*, volume 637 of *Lecture Notes in Computer Sciences*, pages 277–298, September 1992.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection*. Wiley, 1996.

[KID+90] Satoshi Kurihara, Mikio Inari, Norihisa Doi, Kazuki Yasumatsu, and Takemi Yamazaki. SPiCE Collector: The run-time garbage collector for Smalltalk-80 programs translated into C. In *ECOOP/OOPSLA'90 Workshop on Garbage Collection in Object-Oriented Systems*, 1990.

[Mey94] Bertrand Meyer. *Eiffel, The Language*. Prentice Hall, 1994.

[Wil92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *International Workshop on Memory Management (IWMM'92)*, volume 637 of *Lecture Notes in Computer Sciences*, 1992.

[WSNB95] Paul R. Wilson, Mark S.Johnston, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *International Workshop on Memory Management (IWMM'95)*, volume 986 of *Lecture Notes in Computer Sciences*, pages 1–116, 1995.

[ZCC97] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *OOPSLA'97*, volume 32, pages 125–141, October 1997.

[Zor93] Benjamin Zorn. The measured cost of conservative Garbage Collection. *Software Practice and Experience*, 23:733–756, 1993.