

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Marco Túlio Gontijo e Silva

MONOGRAPH OF ORIENTED PROJECT IN COMPUTER SCIENCE I

Implementing the Immix Garbage Collector Algorithm on GHC

Belo Horizonte – MG – Brazil
2010 / 1st semester

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação

**Implementing the Immix Garbage Collector Algorithm on
GHC**

by

Marco Túlio Gontijo e Silva

MONOGRAPH OF ORIENTED PROJECT IN COMPUTER
SCIENCE I

Presented as a requisite for the discipline Oriented Project in Computer Science
of the Computer Science Graduation Course at UFMG

Prof. Dr. Carlos Camarão de Figueiredo

Supervisor

Prof. Dr. Fernando M. Q. Pereira

Co-Supervisor

Belo Horizonte – MG – Brazil

2010 / 1st semester

To my family
I dedicate this work.

Acknowledgements

I would like to thank my parents
for supporting me while I was working on this project,
my wife for the patience,
my supervisor for introducing me to Haskell,
my co-supervisor for helping me in choosing the subject.
This work would not be possible without their help.

“what is the use of a book, thought Alice, without pictures or conversations?”

Lewis Carroll

Contents

List of Figures	vi
List of Acronyms	vii
Abstract	viii
1 INTRODUCTION	9
1.1 Overview	9
1.2 Objective, rationale and motivation	10
2 THEORETICAL REFERENCE	11
2.1 GHC	11
2.1.1 GC Overview	11
2.1.2 Generational GC	11
2.1.3 Last generation	12
2.2 Immix	13
3 METHODOLOGY	14
3.1 Summer of Code	14
3.2 Studies	14
3.3 Discussion	14
3.4 Implementation	15
3.4.1 Liberation in lines	15
4 CONCLUSIONS AND FUTURE WORK	20

Bibliography 21

List of Figures

Figure 2.1	Generational GC	12
Figure 2.2	Allocation on lines	13
Figure 3.1	Current free line representation	19

List of Acronyms

GHC	Glasgow Haskell Compiler
GC	Garbage Collector or Garbage Collection
RTS	Runtime System

Abstract

This monograph describes the work already done on the implementation of the Immix Garbage Collection Algorithm on the Glasgow Haskell Compiler. Immix is reported to be better than the current GC algorithms of GHC: copy collection, mark compact and mark sweep. The first part of the Immix algorithm, freeing memory in units of lines, was implemented. Although the concept of a line is completely new to GHC, few changes were necessary to the source code of GHC.

Keywords: Garbage Collection, Haskell, Immix, Glasgow Haskell Compiler.

1 INTRODUCTION

1.1 Overview

Haskell(1) is one of the most popular functional programming languages. It is widely taught in academia as an example of the functional programming paradigm, and it has seen use in industry(2). Haskell is developed as an open source project, enjoying the support of a very active community of developers.

GHC(3) is the *de facto* standard Haskell compiler(4). This is an open source native code compiler, whose project started at the late eighties. GHC has been mostly developed at the University of Glasgow; however, nowadays it contains patches sent by many different developers, spread all over the world. Currently this compiler is distributed together with the Haskell Platform(5).

Haskell provides developers with managed memory. This means that the running environment shelters the programmers from the difficult and error prone task of manually allocating and freeing memory. In order to deliver this abstraction to developers, Haskell uses a GC(6). Garbage collection is an old field of study in compiler construction, and a good GC implementation is key to performance. In particular, garbage collection has a great impact on the performance of executables produced by GHC(6).

Recently, a new garbage collection algorithm has been published in the Conference on Programming Languages, Design and Implementation (PLDI'08). This new algorithm, so called Immix, was shown to improve on previous state-of-the-art algorithms(7). Immix is perhaps the most modern garbage collection algorithm available in the literature, and this work will investigate whether GHC would benefit from using an implementation of this algorithm.

1.2 Objective, rationale and motivation

The main goal of this work is to implement the Immix GC in the development version of GHC. It will probably result in a better performance for GHC, improving user experience of binaries generated by this compiler, including the XMonad window manager(8) and the revision control system darcs(9).

GHC provides the running application with three garbage collection alternatives for old objects:

- copy collection,
- mark compact collection and
- mark sweep.

Blackburn and McKinley have shown that the Immix GC outperforms all three approaches. In particular, it surpasses the mark sweep implementation in Java benchmarks, with measured performance improvements ranging from 5 to 21%. However, the benefits of Immix have yet to be tested on the running environment of a functional programming language. Contrary to Java, Haskell relies heavily on immutable data. I believe that, as such data tends to have shorter lifetimes than the mutable counter-part, garbage collection suffers more pressure from the running program. This work will investigate whether speculate that, in this new environment, the Immix garbage collector will be particularly useful, as it is expected to nicely handle the high turnover of data.

2 THEORETICAL REFERENCE

2.1 GHC

2.1.1 GC Overview

There are two main techniques for doing GC: reference counting and tracing(10). Reference counting includes, in the object, a counter of references to it. In a tracing GC, there are some objects which are considered as roots of the object graph. Any object accessible via references from these roots is considered alive; the others are garbage, and should be collected. GHC uses a tracing GC.

GHC's memory allocator divides memory in blocks, which makes it easier to

- resize memory areas, by including new blocks on the block list;
- manage large objects, since blocks can be left specially reserved for the purpose of allocation of large objects; and
- free memory, because blocks can be freed individually(11).

2.1.2 Generational GC

GHC's GC is generational, that is, objects are allocated in a generation and, after some time, objects that are still being used are moved to a block in the next generation. This is illustrated by Figure 2.1. Generation 0, called nursery, is subject to Garbage Collection on every run of the GC algorithm. The higher the number of the generation, the lesser times it will be analyzed. This idea assumes that the death probability of younger objects is higher. In general, few objects in a generation are moved to the next one. In a recursive algorithm, where lots of values are allocated and only some of them are used, this strategy achieves a good result. The less allocated values are used, the faster the algorithm runs(12). Measurements indicate that in general the best performance is achieved with 2 generations.

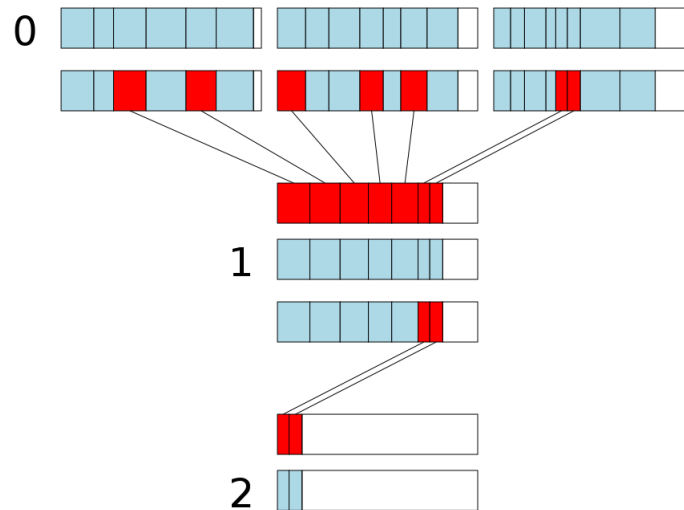


Figure 2.1: Generational GC

2.1.3 Last generation

For the last generation, GHC provides the running application with three garbage collection alternatives:

- *copy collection*,
- *mark compact collection* and
- *mark sweep*.

The choice of which alternative is effectively run is based on the application's relative memory usage: if the application is using less than a certain amount of memory, 30%, copying is used, otherwise GHC uses the mark compact approach(13). Mark sweep is only used when enabled with a flag to the RTS.

Copy collection is similar to the generational algorithm in that live objects are copied to another block. The difference is that the destination block is in the same generation.

In marking algorithms, there is a bitmap of live objects in each block. Each bit in the bitmap corresponds to a word of memory in the block. Only the first bit of each object is marked when it is live. In *mark sweep*, if there are no objects in a block, it is freed. If there are a lot of free spaces in a block, it is considered fragmented, and mark sweep is not used in it again. In *mark compact*, live objects are copied to the same block, using the space liberated by garbage objects.

2.2 Immix

The Immix garbage collector differs from other algorithms, such as mark sweep and mark compact because, unlike these, Immix divides the memory in blocks of the same size, and performs collection on the block level, instead of the object level. Each block, is further divided into same-size lines. Objects may be allocated across lines, as shown in Figure 2.2 (a) and (c), and not always in the beginning of a line, like in (b). It is possible that an object will not fill a line completely, yet experiments(7) show that this extra last space is a small price to pay in exchange for an easier memory layout to be managed.

When a new group of objects needs to be allocated, it is allocated on free lines, as shown in Figure 2.2 (c). As done with mark sweep, only the start of an object is taken into account to see if a line is free or used. Therefore, a line marked as used can have an object starting in it's end — as the last object of line 7 in Figure 2.2 (c) — and using space from the next line, which is not marked as used, since no object starts in it. Because of that, the first line of each group of free lines is not used, and the allocation starts at the beginning of the second line. This technique is called conservative marking, and is proposed by the authors of Immix. It is exemplified in Figure 2.2 (c) by the lines 2, 6 and 8, which were left free in the allocation step. Conservative marking can cause different sizes of unused memory space: bigger than a line, as in line 2; slightly smaller than a line, as in line 6; and much smaller than a line, as in line 8.

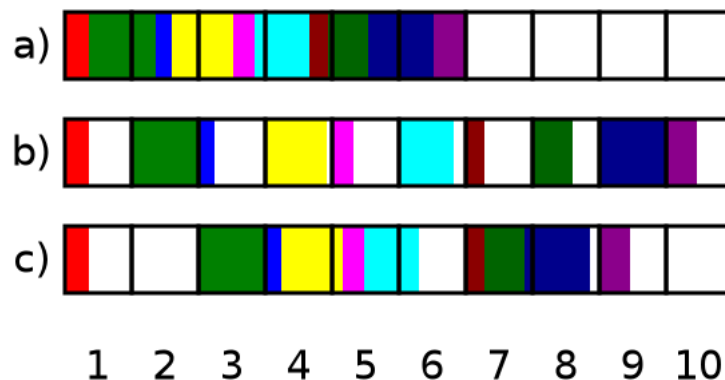


Figure 2.2: Some variations on the allocation on lines: *a)* objects allocated at once, across lines; *b)* objects allocated in the beginning of each line; *c)* conservative marking.

The original Immix algorithm works with objects bigger than a line, but currently we are using other algorithms, like copy collection, in blocks that contains these kind of objects. We're using Immix only for objects smaller than a line.

3 METHODOLOGY

3.1 Summer of Code

I wrote a project to work on this subject in the Google Summer of Code program. This is a program funded by Google that offers stipends to students to enable them to focus on a Free Software project during three months(14). The project must be written by the student on a specific topic and submitted through an organization accepted in the program, and provided with a number of projects. The organization then selects students that will work with them. My project was accepted and now I am working in the program, having been selected by Haskell.org.

3.2 Studies

I have studied Immix in detail using the article(7) and the Technical Report(15). I studied the internals of GHC, specially the RTS, using the source code(16) and the wiki(17). GHC developers use such wiki as a complementary documentation of the source code. The wiki documentation about GHC's GC is known to be outdated, so part of my work was to improve this documentation. I have made some patches of comments and sent them to the GHC repository. They were accepted and are now included in the distribution.

3.3 Discussion

Since May 17th I am publishing updates about my work in my personal weblog(18). I am also discussing any work with Simon Marlow, my Summer of Code mentor, via e-mail(19).

3.4 Implementation

3.4.1 Liberation in lines

The code in `rts/sm/Sweep.c` was simple and similar to what I'm planning to do, so I started changing it. Sweeping in GHC is done by using a bitmap that contains a bit for each word in a memory block and such bit is set to 1 when there's an object starting in the mapped area and 0 otherwise. When there's a block with no objects starting at it, that is, all bits of the bitmap are set to 0, the block is freed.

```
if (resid == 0)
{
    freed++;
    gen->n_old_blocks--;
    if (prev == NULL) {
        gen->old_blocks = next;
    } else {
        prev->link = next;
    }
    freeGroup(bd);
}
```

The bits are analyzed in a group of `BITS_IN(W_)`, where `BITS_IN(W_)` is the number of bits in a word.

```
for (i = 0; i < BLOCK_SIZE_W / BITS_IN(W_); i++)
{
    if (bd->u.bitmap[i] != 0) resid++;
}
```

If more than $\frac{1}{4}$ of the groups are completely set to 0, the block is considered fragmented.

```
if (resid < (BLOCK_SIZE_W * 3) / (BITS_IN(W_) * 4)) {
    fragd++;
    bd->flags |= BF_FRAGMENTED;
}
```

Immix divides blocks of memory in lines. My initial plan was to identify free lines. I decided to consider the size of a line fixed in `BITS_IN(W_)` words, because it will map to a word in the bitmap, and the current implementation already analyzes the blocks in groups of `BITS_IN(W_)` words. This was very easy with the current code.

```
if (bd->u.bitmap[i] != 0) resid++;
else printf("DEBUG: line_found(%p)\n", bd->start + BITS_IN(W_) * i);
```

This worked, and showed some free lines. I'm sure there are other ways of logging in GHC, but `printf` was the simplest way I could thought of. I measured the occurrence of free lines using the `bernouilli` program from the `NoFib` benchmark suite(20), calling it with `500 +RTS -w`, to make it use sweep. In 782 calls to `GarbageCollect()`, `sweep()` was called 171 times, for 41704 blocks to be swept and found 230461 free lines. This gives us about 5.5 free lines per block, from the 8 lines in each block (on 64 bit systems).

The problem is that the bitmap is marked only in the start of the objects allocated, so even in a line with all bits marked with 0 we can't assume that it's completely free, because there may be an object that starts in the previous line that is using the space of the line. Checking only for the previous line does not work either, because a big object can span several lines. What we can do here is a variation of conservative marking, as proposed in (7), checking only the previous line and working only with objects smaller than a line.

To make sure I was working only with objects smaller than a line, I had to mark the blocks that contains medium objects and avoid them when seeking free lines. The block flags are defined in `includes/rts/storage/Block.h`, so I included another flag in this file, `BF_MEDIUM`.

```
/* Block contains objects evacuated during this GC */
#define BF_EVACUATED 1
/* Block is a large object */
#define BF_LARGE 2
/* Block is pinned */
#define BF_PINNED 4
/* Block is to be marked, not copied */
#define BF_MARKED 8
/* Block is free, and on the free list (TODO: is this used?) */
#define BF_FREE 16
/* Block is executable */
```

```

#define BF_EXEC      32
/* Block contains only a small amount of live data */
#define BF_FRAGMENTED 64
/* we know about this block (for finding leaks) */
#define BF_KNOWN     128
/* Block contains objects larger than a line */
#define BF_MEDIUM    256

```

The move of an object from one generation to another is done in the `copy_tag` function of `rts/sm/Evac.c`. I inserted code that checks for the object size and marks the block when it's bigger than `BITS_IN(W_)`.

```

STATIC_INLINE GNUC_ATTR_HOT void
copy_tag(StgClosure **p, const StgInfoTable *info,
         StgClosure *src, nat size, generation *gen, StgWord tag)
{
    StgPtr to, from;
    nat i;

    to = alloc_for_copy(size, gen);

    if(size > BITS_IN(W_)) {
        Bdescr(to)->flags |= BF_MEDIUM;
    }
}

```

So I updated the code in `rts/sm/Sweep.c` to only inspect for free lines in blocks without `BF_MEDIUM` mark.

```

if (bd->u.bitmap[i] != 0) resid++;
else if(!(bd->flags & BF_MEDIUM)) {
    printf("DEBUG: line_found(%p)\n", bd->start + BITS_IN(W_) * i);
}

```

This also worked. Now, in the 32012 blocks there were 189015 free lines, found in the same number of GCs, making about 5.9 free lines per block. We considered only blocks with small objects, but we did not ignore the first line of each group of free lines. This can be achieved by checking if the previous line was also free.

```

if (bd->u.bitmap[i] != 0) resid++;
else if(!(bd->flags & BF_MEDIUM) && i > 0 && bd->u.bitmap[i] == 0) {
    printf("DEBUG: line_found(%p)\n", bd->start + BITS_IN(W_) * i);
}

```

Now, from the 32239 blocks, 165547 free lines were found, giving 5.1 free lines per block. But there are more things to improve. If the whole block is free, we want to free it, instead of marking its lines as free. So it's better to mark the lines after we know that the blocks are not completely free. So I left the code that checks the bitmap as it was, and included a line check only for blocks that are not completely free. At this point, I also associated the fragmentation test with blocks with medium objects, because in blocks of small objects we plan to allocate in free lines, so fragmentation is not a (big) issue.

```

if (resid < (BLOCK_SIZE_W * 3) / (BITS_IN(W_) * 4) &&
    (bd->flags & BF_MEDIUM)) {
    fragd++;
    printf("DEBUG: BF_FRAGMENTED\n");
    bd->flags |= BF_FRAGMENTED;
}
else if(!(bd->flags & BF_MEDIUM)) {
    for(i = 1; i < BLOCK_SIZE_W / BITS_IN(W_); i++)
    {
        if(bd->u.bitmap[i] == 0 && bd->u.bitmap[i - 1] == 0) {
            printf("DEBUG: line_found(%p)\n", bd->start + BITS_IN(W_) * i);
        }
    }
}

```

The total amount of blocks increased dramatically: the blocks that become fragmented and were not called again in sweep made a huge difference. From the 345143 blocks, 1633268 free lines were found, or about 4.7 free lines per block. 9434 blocks were free, so, from the remaining blocks, we have about 4.9 free lines per block.

Something we will need then is a way to access these lines later. The simplest way I thought to achieve it is by constructing a list of lines, where the first word of each free line is a pointer to the next free line, and the first word of the last free line is 0. It's useful to keep reporting the lines to stdout, so that we can then follow the list and check if we are in the same line. A scheme of the list of free lines representation is available in Figure 3.1.



Figure 3.1: Current free line representation

```

if(bd->u.bitmap[i] == 0 && bd->u.bitmap[i - 1] == 0) {
    StgPtr start = bd->start + BITS_IN(W_) * i;
    printf("DEBUG: line_found(%p)\n", start);
    if(line_first == NULL) {
        line_first = start;
    }
    if(line_last != NULL) {
        *line_last = (StgWord) start;
    }
    line_last = start;
    *line_last = 0;
}
}
}
}
}

for(line_last = line_first; line_last; line_last = (StgPtr) *line_last) {
    fprintf(stderr, "DEBUG: line_found(%p)\n", line_last);
}

```

I printed the inclusion of the lines on the list to stdout, and the walking on the list in stderr, so that it'd be easy to spot the differences, specially using the UNIX toll diff. There was no difference between the lists.

4 CONCLUSIONS AND FUTURE WORK

This work has presented the current status of the implementation of a new garbage collection algorithm in the GHC Haskell Compiler. With the simplifications proposed by Simon Marlow, like limiting to objects larger than a line, the implementation of Immix seems to be very simple, and it is likely to result in a gain on performance.

Presently my implementation is able to free memory in lines. In the next step of this project, which will be finished in six months, I will:

- provide a better representation for free lines, that considers groups of lines differently than individual lines;
- implement the algorithm that allocates objects in free lines.

Bibliography

- 1 HASKELL. 2010. [Online; accessed 16-June-2010]. Available from Internet: <<http://haskell.org>>.
- 2 INDUSTRIAL Haskell Group. 2009. [Online; accessed 16-June-2010]. Available from Internet: <<http://industry.haskell.org/>>.
- 3 MARLOW, S. *GHC: The glasgow haskell compiler*. 2010. [Online; accessed 16-June-2010]. Available from Internet: <<http://haskell.org/ghc/>>.
- 4 IMPLEMENTATIONS. 2010. [Online; accessed 16-June-2010]. Available from Internet: <<http://www.haskell.org/haskellwiki/Implementations>>.
- 5 GET the Haskell Platform: The standard haskell development environment. [Online; accessed 16-June-2010]. Available from Internet: <<http://hackage.haskell.org/platform/>>.
- 6 BURRELL, M. *Real-time Haskell?* 2007. [Online; accessed 16-June-2010]. Available from Internet: <<http://mikeburrell.wordpress.com/2007/02/01/real-time-haskell/>>.
- 7 BLACKBURN, S. M.; MCKINLEY, K. S. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. *ACM Conference on Programming Language Design and Implementation*, p. 22–32, 2008.
- 8 XMONAD: That was easy, xmonad rocks! [Online; accessed 16-June-2010]. Available from Internet: <<http://xmonad.org/>>.
- 9 DARCS: Distributed. interactive. smart. 2010. [Online; accessed 16-June-2010]. Available from Internet: <<http://darcs.net/>>.
- 10 WIKIPEDIA. *Garbage collection (computer science) — Wikipedia, The Free Encyclopedia*. 2010. [Online; accessed 16-June-2010]. Available from Internet: <[http://en.wikipedia.org/wiki/Garbage_collection_\(computer_science\)](http://en.wikipedia.org/wiki/Garbage_collection_(computer_science))>.
- 11 THE Block Allocator. [Online; accessed 16-June-2010]. Available from Internet: <<http://hackage.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/BlockAlloc>>.
- 12 MARLOW, S. *GHC/Memory Management*. [Online; accessed 16-June-2010]. Available from Internet: <http://www.haskell.org/haskellwiki/GHC/Memory_Management>.
- 13 JAMES, R. *The GHC Garbage Collector Notes*. 2006. [Online; accessed 16-June-2010]. Available from Internet: <<http://hackage.haskell.org/trac/ghc/wiki/GarbageCollectorNotes>>.
- 14 GOOGLE. *Google Summer of Code*. 2010. [Online; accessed 17-June-2010]. Available from Internet: <<http://code.google.com/soc/>>.

- 15 BLACKBURN, S. M.; MCKINLEY, K. S. *Immix garbage collection: Fast collection, space efficiency, and mutator locality*. Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University, Canberra, 2007.
- 16 INDEX of /ghc. 2010. [Online; accessed 17-June-2010]. Available from Internet: <<http://darcs.haskell.org/ghc/>>.
- 17 THE GHC Commentary. [Online; accessed 17-June-2010]. Available from Internet: <<http://hackage.haskell.org/trac/ghc/wiki/Commentary>>.
- 18 BLOG do Marcot. 2010. [Online; accessed 17-June-2010]. Available from Internet: <<http://marcotmarcot.wordpress.com/>>.
- 19 SILVA, M. T. G. e. *Some questions about GHC source code and wiki*. 2010. [Online; accessed 17-June-2010]. Available from Internet: <<http://www.haskell.org/pipermail/cvs-ghc/2010-May/054037.html>>.
- 20 JONES, S. P. *The NoFib benchmark suite*. 1997. [Online; accessed 17-June-2010]. Available from Internet: <<http://www.dcs.gla.ac.uk/fp/software/ghc/nofib.html>>.