New Allocator Manual

Version 0.1, March 2005

Marcus Crestani

1 Introduction

This manual provides information about the new allocator, called 'mc-alloc'. It describes the changes and the rationale to replace the old XEmacs allocator. This document will once be merged into the XEmacs Internals Manual.

The allocator, which was in XEmacs for a couple of years and which is still used by default, is referred to as the "old allocator". The newly written allocator, which can be enabled configuring with '`--enable-mc-alloc`', is called the "new allocator" in the following.

This document and the new allocator were written by Marcus Crestani.

2 Basics

An allocator is part of a memory manager. It administrates free memory and serves memory needs of programs. Management of free memory is difficult, since memory usually gets fragmented during execution of a program, and because memory is allocated and freed frequently. In an automatically managed heap, the freeing is usually done by a garbage collector.

To keep fragmentation minimal, an allocator has to find the best fitting spot where to allocate a new object as fast as possible. This can be done by maintaining a data structure called free list. It contains free contiguous blocks of memory of a certain size. If memory is requested, the free list is traversed to find a fitting block of memory.

First, a few words about the old XEmacs allocator. The old allocator has several ways to handle different kinds of Lisp objects. Each object is allocated by using its own free list algorithm:

- Simple data types, whose values directly represent the contents of the Lisp object (i.e. integers and characters). These objects are not managed by the allocator because they do not occupy space in the heap.
- *Lrecords* are rather small Lisp objects. They are allocated on the heap. According to their type they are allocated in FROB blocks. This includes the objects that are most common and relatively small (i.e. cons cells, strings, subrs, floats, compiled functions, symbols, extents, events, and markers). Lrecords of one type have the same size. Thus, they can be held in fix-sized blocks. The old allocator maintains data structures for each object's fixed sized blocks, called FROB blocks.
- *Lcrecords* are also allocated on the heap, but they are individually `malloc()`ed, because they have variable sizes. Usually, less frequently used and bigger objects are classified lcrecords. The old allocator does not really manage those kinds of Lisp objects, it passes the allocation to the system `malloc`, which is not maximally efficient.
- There are a couple of special case allocations:
  - Strings are allocated in two parts, a fix-size object (containing the length, property list, and a pointer to the actual data) as a lrecord in frob blocks, whereas the actual string data is allocated separately. For short strings, the string data is kept in so called string-chars blocks, which get compactified (and thus relocated) during garbage collection. Big strings are individually `malloc()`ed.
  - Buffers are also allocated in two parts. The maintainance information about the buffer is kept in a lcrecord, whereas the buffer content is allocated by the relocating allocator (where available). The relocating allocator does some optimizations to make sure, free space is faster returned to the operating system.
  - ###TODO###: There are some more special case objects to be listed here.

For more detailed information see section "Allocation of Objects in XEmacs Lisp" in *XEmacs Internals Manual*.

Other drawbacks of the old allocator are:

- The allocation of lrecords is tightly coupled with the memory manager. It uses a lot of macros. Therefore the definition of some basic Lisp objects has to be made in `alloc.c`.
- The finalization of lrecords has a very poor macro interface.

- The mark bits are tucked into lrecords and lcreords, which generally leads to bad locality during mark phase, and therefore to bad caching efficiency. This is bad for performance.

All these special cases and drawbacks lead to a large and complex allocator code. This goes hand in hand with a complicated interface. Thus, the code of the old allocator is very hard to maintain.

The final goal of all the work done in the area of memory management is to replace XEmacs's primitive garbage collector. Therefore a clean and straight forward interface to the allocator is needed.

Writing the new allocator was not the hard part of the work. The problem was to hook it into XEmacs and identify all special cases and clean up all the allocator-dependent code. This also led to modifications to the portable dumper and the garbage collection algorithms (without improving them).

In Chapter 3 [The New Allocator], page 5, I describe the functionality and the basic algorithms of the new allocator. Very usefull might be Section 3.12 [Glossary], page 12. To get a quick overview about the new allocator's inteface, see Section 3.13 [Interface to the New Allocator], page 12.

The changes to the XEmacs sources are described in Chapter 4 [Changes to XEmacs], page 15.

You can find a list with unresolved issues in Chapter 5 [TODO], page 35.

To document the current state of the work, I did some benchmarking. See Chapter 6 [Benchmarks], page 37.

Please keep in mind that the new allocator is work in progress!

3 The New Allocator

The ideas and algorithms are based on the allocator of the *Boehm-Demers-Weiser conservative garbage collector*. See `http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html`.

3.1 Three-Level Allocation

The new allocator efficiently manages the allocation of Lisp objects by minimizing the number of times `malloc()` and `free()` are called. The allocation process has three layers of abstraction:

1. It allocates memory in very large chunks called *heap sections*.
2. The heap sections are subdivided into *pages*. The *page size* is determined by the constant `PAGE_SIZE`. It holds the size of a page in bytes.
3. One page consists of one or more *cells*. Each cell represents a memory location for an object. The cells on one page all have the same size, thus every page only contains equal-sized objects.

If an object is bigger than page size, it is allocated on a *multi-page*. Then there is only one cell on a multi-page (the cell covers the full multi-page). Is an object smaller than $\frac{1}{2}$ `PAGE_SIZE`, a page contains several objects and several cells. There is only one cell on a page for object sizes from $\frac{1}{2}$ `PAGE_SIZE` to `PAGE_SIZE` (whereas multi-pages always contain only one cell).

Only in layer one `malloc()` and `free()` are called.

3.2 Size Classes and Page Lists

Meta-information about every page and multi-page is kept in a *page header*. The page header contains some bookkeeping information like number of used and free cells, and pointers to other page headers. The page headers are linked in a *page list*.

Every page list builds a *size class*. A size class contains all pages (linked via page headers) for objects of the same size. The new allocator does not group objects based on their type, it groups objects based on their sizes.

Here is an example: A `cons` contains a `lrecord_header`, a `car` and `cdr` field. Altogether it uses 12 bytes of memory (on 32 bits machines). All conses are allocated on pages with a cell size of 12 bytes. All theses pages are kept together in a page list, which represents the size class for 12 bytes objects. But this size class is not exclusively for conses only. Other objects, which are also 12 bytes big (e.g. weak-boxes), are allocated in the same size class and on the same pages.

The number of size classes is customizable, so is the size step between successive size classes.

3.3 Used and Unused Heap

The memory which is managed by the allocator can be divided in two logical parts:

The *used heap* contains pages, on which objects are allocated. These pages are completely or partially occupied. In the used heap, it is important to quickly find a free spot for a new object. Therefore the size classes of the used heap are defined by the size of the cells on the pages. The size classes should match common object sizes, to avoid wasting memory.

The *unused heap* only contains completely empty pages. They have never been used or have been freed completely again. In the unused heap, the size of consecutive memory tips the scales. A page is the smallest entity which is asked for. Therefore, the size classes of the unused heap are defined by the number of consecutive pages.

The parameters for the different size classes can be adjusted independently, see Section 3.3.1 [Adjust Size Classes of the Used Heap], page 6 and Section 3.3.2 [Adjust Size Classes of the Unused Heap], page 6.

3.3.1 Adjust Size Classes of the Used Heap

Adjust the size classes in `mc-alloc.h`:

```
/* Heap used list constants: In the used heap, it is important to
   quickly find a free spot for a new object. Therefore the size
   classes of the used heap are defined by the size of the cells on
   the pages. The size classes should match common object sizes, to
   avoid wasting memory. */

/* Minimum object size in bytes. */
#define USED_LIST_MIN_OBJECT_SIZE    8

/* The step size by which the size classes increase (up to upper
   threshold). This many bytes are mapped to a single used list: */
#define USED_LIST_LIN_STEP           4

/* The upper threshold should always be set to PAGE_SIZE/2, because if
   a object is larger than PAGE_SIZE/2 there is no room for any other
   object on this page. Objects this big are kept in the page list of
   the multi-pages, since a quick search for free spots is not
   needed for this kind of pages (because there are no free spots).
   PAGE_SIZES_DIV_2 defines maximum size of a used space list. */
#define USED_LIST_UPPER_THRESHOLD PAGE_SIZE_DIV_2
```

3.3.2 Adjust Size Classes of the Unused Heap

Adjust the size classes in `mc-alloc.h`:

```
/* Heap free list constants: In the unused heap, the size of
   consecutive memory tips the scales. A page is smallest entity which
   is asked for. Therefore, the size classes of the unused heap are
   defined by the number of consecutive pages. */

/* Sizes up to this many pages each have their own free list. */
#define FREE_LIST_LOWER_THRESHOLD   32
/* The step size by which the size classes increase (up to upper
   threshold). FREE_LIST_LIN_STEP number of sizes are mapped to a
   single free list for sizes between FREE_LIST_LOWER_THRESHOLD and
   FREE_LIST_UPPER_THRESHOLD. */
#define FREE_LIST_LIN_STEP 8
/* Sizes of at least this many pages are mapped to a single free
   list. Blocks of memory larger than this number are all kept in a
   single list, which makes searching this list slow. But objects that
   big are really seldom. */
#define FREE_LIST_UPPER_THRESHOLD  256
```

3.4 Mapping of Heap Pointers to Page Headers

For caching benefits, the page headers and mark bits are stored separately from their associated page. During garbage collection (i.e. for marking and freeing objects) it is important to identify the page header which is responsible for a given Lisp object.

To do this task quickly, I added a two level search tree: the upper 10 bits of the heap pointer are the index of the first level. This entry of the first level links to the second level, where the next 10 bits of the heap pointer are used to identify the page header. The remaining bits point to the object relative to the page.

On architectures with more than 32 bits pointers, a hash value of the upper bits is used to index into the first level.

3.5 Mark Bits

For caching purposes, the mark bits are no longer kept within the objects, they are kept in a separate bit field.

Every page header has a field for the mark bits of the objects on the page. If there are less cells on the page than there fit bits in the integral data type `EMACS_INT`, the mark bits are stored directly in this `EMACS_INT`.

Otherwise, the mark bits are written in a separate space, with the page header pointing to this space. This happens to pages with rather small objects: many cells fit on a page, thus many mark bits are needed.

Use the following functions/macros:

```
void set_mark_bit (void *ptr, EMACS_INT value)
  Set the mark bit of the object pointed to by ptr to value.

EMACS_INT get_mark_bit (void *ptr)
  Return the mark bit of the object pointed to by ptr.
```

```
bool MARKED_P(ptr)
  [MACRO] Returns true if the mark bit of the object pointed to by ptr
  is set.

MARK(ptr)
  [MACRO] Marks the object pointed to by ptr (sets the mark bit to 1).

UNMARK(ptr)
  [MACRO] Unmarks the object pointed to by ptr (sets the mark bit to 0).
```

3.6 Allocate Memory

Use

```
void *mc_alloc (size_t size)
  Returns a pointer to a newly allocated block of memory of given size.
```

to request memory from the allocator.

This is how the new allocator allocates memory:

1. Determine the size class of the object.
2. Is there already a page in this size class and is there a free cell on this page?
   - YES
     3. Unlink free cell from free list, return address of free cell. DONE.
   - NO
     3. Is there a page in the unused heap?
        - YES
          4. Move unused page to used heap.
          5. Initialize page header, free list, and mark bits.
          6. Unlink first cell from free list, return address of cell. DONE.
        - NO
          4. Expand the heap, add new memory to unused heap [go back to 3. and proceed with the YES case].

The allocator puts partially filled pages to the front of the page list, completely filled ones to the end. That guarantees a fast terminating search for free cells. Are there two successive full pages at the front of the page list, the complete size class is full, a new page has to be added.

3.7 Expand Heap

To expand the heap, a big chunk of contiguous memory is allocated using `malloc()`. These pieces are called heap sections. How big a new heap section is (and thus the growth of the heap) is adjustable:

```
/* Heap growth constants. Heap increases by any number between the
   boundaries (unit is PAGE_SIZE). */
#define MIN_HEAP_INCREASE          32
#define MAX_HEAP_INCREASE         256 /* not used */

/* Every heap growth is calculated like this:
   needed_pages + ( HEAP_SIZE / ( PAGE_SIZE * HEAP_GROWTH_DIVISOR )).
   So the growth of the heap is influenced by the current size of the
   heap, but kept between MIN_HEAP_INCREASE and MAX_HEAP_INCREASE
   boundaries.
   This reduces the number of heap sectors, the larger the heap grows
   the larger are the newly allocated chunks. */
#define HEAP_GROWTH_DIVISOR         3
```

This approach keeps the number of heap sections small: the bigger the heap grows, the bigger the heap sections get. If all pages of a heap section are freed, the complete heap section is returned to the operating system by using `free()`.

3.8 Free Memory

One optimization in XEmacs is that locally used Lisp objects are freed manually (the memory is not wasted till the next garbage collection). Therefore the new allocator provides this function:

```
void mc_free (void *ptr)
  Frees the object pointed to by ptr.
```

This function is also used internally during sweep phase of the garbage collection.

This is how it works in detail:

1. Use pointer to identify page header (use lookup mechanism described in Section 3.4 [Mapping of Heap Pointers to Page Headers], page 7).
2. Mark cell as free and hook it into free list.
3. Is the page completely empty?
   - YES
     4. Unlink page from page list.
     5. Remove page header, free list, and mark bits.
     6. Move page to unused heap.
   - NO
     4. Move page to front of size class (to speed up allocation of objects).

If the last object of a page is freed, the empty page is returned to the unused heap. The allocator tries to coalesce adjacent pages, to gain a big piece of contiguous memory. The resulting chunk is hooked into the according size class of the unused heap. If this created a complete heap section, the heap section is returned to the operating system by using `free()`.

3.9 Allocator and Garbage Collector

The different way the new allocator handles Lisp objects lead to some changes in the garbage collector. These changes are not improving the algorithms significantly, but they are simplifying the interface and thus provide a good basis for any future garbage collector changes.

In the following, the additional functions provided by the allocator especially for garbage collection are described. The changes to the XEmacs sources caused by these functions are described in Chapter 4 [Changes to XEmacs], page 15.

3.9.1 Allocator and Finalization

The new allocator provides finalization mechanisms to be used by the garbage collector. The provided functions are described here:

```
void *mc_finalize (void)
  Runs MC_ALLOC_CALL_FINALIZER on all unmarked objects in the used heap.
```

The finalizer of every unmarked object is called. The macro `MC_ALLOC_CALL_FINALIZER` has to be defined and call the finalizer of the object. For Lisp objects it looks like this:

```
/* Tell mc-alloc how to call a finalizer. */
#define MC_ALLOC_CALL_FINALIZER(ptr)                                    \
{                                                                       \
  Lisp_Object MCACF_obj = wrap_pointer_1 (ptr);                         \
  struct lrecord_header *MCACF_lheader = XRECORD_LHEADER (MCACF_obj);   \
  if (XRECORD_LHEADER (MCACF_obj) && LRECORDP (MCACF_obj)               \
      && !LRECORD_FREE_P (MCACF_lheader)  )                             \
    {                                                                   \
      const struct lrecord_implementation *MCACF_implementation         \
        = LHEADER_IMPLEMENTATION (MCACF_lheader);                       \
      if (MCACF_implementation && MCACF_implementation->finalizer)      \
        MCACF_implementation->finalizer (ptr, 0);                       \
    }                                                                   \
} while (0)
```

3.9.2 Sweep Phase

With the new allocator, I was able to simplify the sweep phase significantly: sweep functions for every single lrecord are no longer needed. A call to

```
void* mc_sweep (void)
  Frees all unmarked objects in the used heap.
```

does all the work for the garbage collector. It visits all used pages and frees all the umarked objects.

3.10 Allocator and Dumper

The new allocator provides the following functionality for the dumper:

```
    void *mc_finalize_for_disksave (void)
      Runs MC_ALLOC_CALL_FINALIZER_FOR_DISKSAVE on all objects in the
      used heap.
```

The finalizer for disksave of every object is called to shrink the dump image. The finalizer for disksave is applied on every object in the used heap. The macro `MC_ALLOC_CALL_FINALIZER_FOR_DISKSAVE` has to be defined and call the finalizer for disksave of the object. For Lisp objects it looks like this:

```
/* Tell mc-alloc how to call a finalizer for disksave. */
#define MC_ALLOC_CALL_FINALIZER_FOR_DISKSAVE(ptr)                         \
{                                                                         \
  Lisp_Object MCACF_obj = wrap_pointer_1 (ptr);                           \
  struct lrecord_header *MCACF_lheader = XRECORD_LHEADER (MCACF_obj);     \
  if (XRECORD_LHEADER (MCACF_obj) && LRECORDP (MCACF_obj)                 \
      && !LRECORD_FREE_P (MCACF_lheader)  )                               \
    {                                                                     \
      const struct lrecord_implementation *MCACF_implementation           \
        = LHEADER_IMPLEMENTATION (MCACF_lheader);                         \
      if (MCACF_implementation && MCACF_implementation->finalizer)        \
        MCACF_implementation->finalizer (ptr, 1);                         \
    }                                                                     \
} while (0)
```

3.11 Unmanaged Heap

Note: This is a new feature of the allocator, not fully implemented and used yet!

Like used heap, but not managed by the garbage collector. Objects are allocated the same way, but have to be freed manually. The unmanaged heap is not touched by the sweep function.

The rationale for the unmanaged heap is: Other data structures than Lisp objects can be allocated with the new allocator on the unmanaged heap. So they would use the same efficient abstractions of the new allocater and the calls to `malloc` and `free` would be minimized.

###TODO###: Further examinations and measurements are needed. if there is a performance benefit, all XEmacs dynamic data structures should probably be allocated that way.

See Chapter 5 [TODO], page 35: Strings may be a good candidate to be partially allocated on the unmanaged heap.

```
    void *mc_alloc_unmanaged (size_t size); Returns a pointer to a block of
    memory of given size on the unmanaged heap.

    void *mc_realloc_unmanaged (void *ptr, size_t size);
```

```
Modifies the size of the memory block pointed to by ptr. The
Address of the new block of given size is returned.
```

3.12 Glossary

`PAGE_SIZE`	Adjustable size of one page
heap section	Big chunk of contiguous memory, which is obtained from the operating system.
page	A piece of memory (the size of `PAGE_SIZE`). Used for allocating objects, which are smaller then `PAGE_SIZE`. The memory for a page is taken from a heap section.
multi-page	Consists of several contiguous `PAGE_SIZE`-sized pieces memory, for objects larger than `PAGE_SIZE`.
cell	Storage location for one objects. Depending on the size of the object, there can be many cells on one page.
object	Data structure, which needs to be allocated by the memory manager. Every object is stored in a cell on a page.
size class	One size class contains all objects of one size. Every size class is associated with a page list.
page list	Links all pages with same cell sizes via their page headers.
page header	Contains meta information about the page and the object stored on the page (like statistics and mark bits).
heap	Area in memory where all the objects are allocated.
used heap	Contains pages, on which objects are allocated. These pages are completely or partially occupied.
unused heap	Contains completely empty pages. They have never been used or have been freed completely again.
unmanaged heap	Like used heap, but not managed by the garbage collector. Objects are allocated the same way, but have to be freed manually.

3.13 Interface to the New Allocator

- Allocation related functions and macros:

  `void` **init_mc_allocator** (`void`) [Function]

  Builds and initializes all needed datastructures of the new allocator.

`void *` **mc_alloc** (`size_t size`) [Function]
: Returns a pointer to a newly allocated block of memory of given `size` on the used heap.

`void` **mc_free** (`void *ptr`) [Function]
: Frees the object pointed to by `ptr`.

`void *` **mc_realloc** (`void *ptr, size_t size`) [Function]
: Modifies the size of the memory block pointed to by `ptr`. The Address of the new block of given `size` is returned.

- Garbage collection related functions and macros:

`void` **set_mark_bit** (`void *ptr, EMACS_INT value`) [Function]
: Set the mark bit of the object pointed to by `ptr` to `value`.

`EMACS_INT` **get_mark_bit** (`void *ptr`) [Function]
: Return the mark bit of the object pointed to by `ptr`.

`EMACS_INT` **MARKED_P** (`ptr`) [Macro]
: Returns true if the mark bit of the object pointed to by `ptr` is set.

MARK (`ptr`) [Macro]
: Marks the object pointed to by `ptr` (sets the mark bit to 1).

UNMARK (`ptr`) [Macro]
: Unmarks the object pointed to by `ptr` (sets the mark bit to 0).

`void *` **mc_finalize** (`void`) [Function]
: The finalizer of every not marked object is called. The macro `MC_ALLOC_CALL_FINALIZER` has to be defined and call the finalizer of the object.

`void *` **mc_sweep** (`void`) [Function]
: All not marked objects of the used heap are freed.

- Portable dumper related functions and macros:

`void *` **mc_finalize_for_disksave** (`void`) [Function]
: The finalizer for disksave of every object is called to shrink the dump image. The macro `MC_ALLOC_CALL_FINALIZER_FOR_DISKSAVE` has to be defined and call the finalizer for disksave of the object.

- Functions and macros related with allocation statistics:

`Bytecount` **mc_alloced_storage_size** (`Bytecount claimed_size, struct overhead_stats *stats`) [Function]
: Returns the real size, including overhead, which is actually allocated for an object with given `claimed_size`.

mc-alloc-memory-usage [Lisp-Function]
: Returns stats about the mc-alloc memory usage. See `diagnose.el`.

show-mc-alloc-memory-usage [Lisp-Function]
: Pretty prints stats about the mc-alloc memory usage. See `diagnose.el`.

- Allocation function for the unmanaged heap:

  `void *` **mc_alloc_unmanaged** (`size_t size`) [Function]

  Returns a pointer to a newly allocated block of memory of given `size` on the unmanaged heap.

  `void *` **mc_realloc_unmanaged** (`void *ptr, size_t size`) [Function]

  Modifies the size of the memory block pointed to by `ptr`. The Address of the new block of given `size` is returned.

4 Changes to XEmacs

In this chapter I am explaining all changes I made to the XEmacs sources to add the new allocator. In every section the according ChangeLog entries are listed.

4.1 New configure flag for `MC_ALLOC`

I added a new configure flag '`--enable-mc-alloc`' ('`--mc-alloc`' for autoconf 2.13) for enabling the new allocator.

```
ChangeLog addition:

        New configure flag: 'MC_ALLOC':

        * configure.ac (XE_COMPLEX_ARG_ENABLE): Add '--enable-mc-alloc' as
        a new configure flag.
        * configure.in (AC_INIT_PARSE_ARGS): Add '--mc-alloc' as a new
        configure flag.
        * configure.usage: Add description for 'mc-alloc'.

src/ChangeLog addition:

        New configure flag: 'MC_ALLOC':

        * config.h.in: Add new flag 'MC_ALLOC'.

nt/ChangeLog addition:

        New configure flag: 'MC_ALLOC':

        * config.inc.samp: Add new flag 'MC_ALLOC'.
        * xemacs.mak: Add flag and configuration output for 'MC_ALLOC'.
```

4.2 New files

I added two new files:

1. `src/mc-alloc.c`
2. `src/mc-alloc.h`

```
src/ChangeLog addition:

        New files:

        * Makefile.in.in: Add new object file mc-alloc.o.
        * depend: Add new files to dependencies.
        * mc-alloc.c: New.
        * mc-alloc.h: New.
```

```
nt/ChangeLog addition:

        New files:

        * xemacs.dsp: Add source files mc-alloc.c and mc-alloc.h.
        * xemacs.mak: Add new object file mc-alloc.obj to dependencies.
```

4.3 Plugging the new allocator into XEmacs

To get the new allocator running, a few changes and code movements were needed to initialize it correctly.

```
src/ChangeLog addition:

        Running the new allocator from XEmacs:

        * alloc.c (deadbeef_memory): Moved to mc-alloc.c.
        * emacs.c (main_1): Initialize the new allocator and add
        syms_of_mc_alloc.
        * symsinit.h: Add syms_of_mc_alloc.
```

4.4 Remove old lrecord FROB block allocation

The first step is to replace the FROB block allocation with the new allocator.

I created new lrecord allocation functions:

- If the size of the lrecord is fix, say it equals its size of its struct, then use `alloc_lrecord_type`.
- If the size varies, i.e. it is not equal to the size of its struct, use `alloc_lrecord` and specify the amount of storage you need for the object.
- Some lrecords, which are used totally internally, use the `noseeum_alloc_lrecord` function for the reason of debugging.
- To free a Lisp_Object manually, use `free_lrecord`.

The allocation functions return a pointer to a storage location for the new object with a fully initialized `lrecord_header`.

Handing all the lrecords to the new allocator leads to changes in the way they are finalized and sweeped (see Section 4.5 [Lrecord finalizer], page 19).

Additionally, the quite old `breathing_space` functionality, which always holds a small amount of memory, which is released if memory is running out, is removed. I think this feature is not needed anymore today.

```
src/ChangeLog addition:

        New lrecord allocation and free functions:
```

```
* alloc.c (alloc_lrecord): New. Allocates an lrecord, includes
type checking and initializing of the lrecord_header.
* alloc.c (noseeum_alloc_lrecord): Same as above, but increments
the NOSEEUM cons counter.
* alloc.c (free_lrecord): New. Calls the finalizer and frees the
lrecord.
* lrecord.h: Add lrecord allocation prototypes and comments.

Remove old lrecord FROB block allocation:

* alloc.c (allocate_lisp_storage): Former function to expand
heap. Not needed anymore, remove.
* alloc.c: Completely remove ‘Fixed-size type macros’
* alloc.c (release_breathing_space): Remove.
* alloc.c (memory_full): Remove release_breathing_space.
* alloc.c (refill_memory_reserve): Remove.
* alloc.c (TYPE_ALLOC_SIZE): Remove.
* alloc.c (DECLARE_FIXED_TYPE_ALLOC): Remove.
* alloc.c (ALLOCATE_FIXED_TYPE_FROM_BLOCK): Remove.
* alloc.c (ALLOCATE_FIXED_TYPE_1): Remove.
* alloc.c (ALLOCATE_FIXED_TYPE): Remove.
* alloc.c (NOSEEUM_ALLOCATE_FIXED_TYPE): Remove.
* alloc.c (struct Lisp_Free): Remove.
* alloc.c (LRECORD_FREE_P): Remove.
* alloc.c (MARK_LRECORD_AS_FREE): Remove.
* alloc.c (MARK_LRECORD_AS_NOT_FREE): Remove.
* alloc.c (PUT_FIXED_TYPE_ON_FREE_LIST): Remove.
* alloc.c (FREE_FIXED_TYPE): Remove.
* alloc.c (FREE_FIXED_TYPE_WHEN_NOT_IN_GC): Remove.

Allocate old lrecords with new allocator:

* alloc.c: DECLARE_FIXED_TYPE_ALLOC removed for all lrecords
defined in alloc.c.
* alloc.c (Fcons): Allocate with new allocator.
* alloc.c (noseeum_cons): Allocate with new allocator.
* alloc.c (make_float): Allocate with new allocator.
* alloc.c (make_bignum): Allocate with new allocator.
* alloc.c (make_bignum_bg): Allocate with new allocator.
* alloc.c (make_ratio): Allocate with new allocator.
* alloc.c (make_ratio_bg): Allocate with new allocator.
* alloc.c (make_ratio_rt): Allocate with new allocator.
* alloc.c (make_bigfloat): Allocate with new allocator.
* alloc.c (make_bigfloat_bf): Allocate with new allocator.
* alloc.c (make_compiled_function): Allocate with new allocator.
* alloc.c (Fmake_symbol): Allocate with new allocator.
* alloc.c (allocate_extent): Allocate with new allocator.
* alloc.c (allocate_event): Allocate with new allocator.
* alloc.c (make_key_data): Allocate with new allocator.
* alloc.c (make_button_data): Allocate with new allocator.
* alloc.c (make_motion_data): Allocate with new allocator.
* alloc.c (make_process_data): Allocate with new allocator.
* alloc.c (make_timeout_data): Allocate with new allocator.
* alloc.c (make_magic_data): Allocate with new allocator.
* alloc.c (make_magic_eval_data): Allocate with new allocator.
* alloc.c (make_eval_data): Allocate with new allocator.
* alloc.c (make_misc_user_data): Allocate with new allocator.
* alloc.c (Fmake_marker): Allocate with new allocator.
* alloc.c (noseeum_make_marker): Allocate with new allocator.
* alloc.c (make_uninit_string): Allocate with new allocator.
* alloc.c (resize_string): Allocate with new allocator.
* alloc.c (make_string_nocopy): Allocate with new allocator.

Garbage Collection:

* alloc.c (GC_CHECK_NOT_FREE): Remove obsolete assertions.
* alloc.c (SWEEP_FIXED_TYPE_BLOCK): Remove.
* alloc.c (SWEEP_FIXED_TYPE_BLOCK_1): Remove.
* alloc.c (sweep_conses): Remove.
* alloc.c (free_cons): Use new allocator to free.
* alloc.c (sweep_compiled_functions): Remove.
* alloc.c (sweep_floats): Remove.
* alloc.c (sweep_bignums): Remove.
* alloc.c (sweep_ratios): Remove.
* alloc.c (sweep_bigfloats): Remove.
* alloc.c (sweep_symbols): Remove.
* alloc.c (sweep_extents): Remove.
* alloc.c (sweep_events): Remove.
* alloc.c (sweep_key_data): Remove.
* alloc.c (free_key_data): Use new allocator to free.
* alloc.c (sweep_button_data): Remove.
* alloc.c (free_button_data): Use new allocator to free.
* alloc.c (sweep_motion_data): Remove.
* alloc.c (free_motion_data): Use new allocator to free.
* alloc.c (sweep_process_data): Remove.
* alloc.c (free_process_data): Use new allocator to free.
* alloc.c (sweep_timeout_data): Remove.
* alloc.c (free_timeout_data): Use new allocator to free.
* alloc.c (sweep_magic_data): Remove.
* alloc.c (free_magic_data): Use new allocator to free.
* alloc.c (sweep_magic_eval_data): Remove.
* alloc.c (free_magic_eval_data): Use new allocator to free.
* alloc.c (sweep_eval_data): Remove.
* alloc.c (free_eval_data): Use new allocator to free.
```

```
        * alloc.c (sweep_misc_user_data): Remove.
        * alloc.c (free_misc_user_data): Use new allocator to free.
        * alloc.c (sweep_markers): Remove.
        * alloc.c (free_marker): Use new allocator to free.
        * alloc.c (garbage_collect_1): Remove release_breathing_space.
        * alloc.c (gc_sweep): Remove all the old lcrecord and lrecord
        related stuff. Sweeping now works like this: compact string
        chars, finalize, sweep.
        * alloc.c (common_init_alloc_early): Remove old lrecord
        initializations, remove breathing_space.
        * emacs.c (Fdump_emacs): Remove release_breathing_space.
        * lisp.h: Remove prototype for release_breathing_space.
        * lisp.h: Adjust the special cons mark makros.
```

4.5 Lrecord finalizer

Finalization for lrecords was done by the `ADDITIONAL_FREE_*` macros. This was a rather poor way doing it and was tightly coupled with the garbage collector. Lrecords now have real finalization functions. I transformed the macros to functions and added them to the lrecord definition.

Strings are more special: for more information about strings, see Section 4.10 [Strings], page 28.

```
src/ChangeLog addition:

        Lrecord Finalizer:

        * alloc.c: Add finalizer to lrecord definition.
        * alloc.c (finalize_string): Add finalizer for string.
        * bytecode.c: Add finalizer to lrecord definition.
        * bytecode.c (finalize_compiled_function): Add finalizer for
        compiled function.
        * marker.c: Add finalizer to lrecord definition.
        * marker.c (finalize_marker): Add finalizer for marker.

        These changes build the interface to mc-alloc:

        * lrecord.h (MC_ALLOC_CALL_FINALIZER): Tell mc-alloc how to
        finalize lrecords.
        * lrecord.h (MC_ALLOC_CALL_FINALIZER_FOR_DISKSAVE): Tell
        mc-alloc how to finalize for disksave.
```

4.6 Unify lrecords and lcrecords

The distinction between lrecords and lcrecords, which was justified only by their different ways of allocation, is not needed any longer. Thus, out of the two different old `lrecord_header` and `lcrecord_header` I made one new `lrecord_header`:

```
struct lrecord_header
{
  /* Index into lrecord_implementations_table[].  Objects that have been
     explicitly freed using e.g. free_cons() have lrecord_type_free in
     this field. */
  unsigned int type :8;

  /* 1 if the object is readonly from lisp */
  unsigned int lisp_readonly :1;

  /* The ‘free’ field is a flag that indicates whether this lrecord
     is currently free or not. This is used for error checking and
     debugging. */
  unsigned int free :1;

  /* The ‘uid’ field is just for debugging/printing convenience.
     Having this slot doesn’t hurt us much spacewise, since the
     bits are unused anyway. */
  unsigned int uid :22;
}
```

Note: the mark bits are removed from the `lrecord_header`, the new allocator keeps track of the mark bits in a separate location.

Some functions conditioning on lrecord or lcrecord could be simplified. Also, I adjusted the comments to reflect the new situation.

```
src/ChangeLog addition:

        Unify lrecords and lcrecords:

        * lisp.h (struct Lisp_String): Adjust string union hack to
        new lrecord header.
        * lrecord.h: Adjust comments.
        * lrecord.h (struct lrecord_header): The new lrecord header
        includes type, lisp-readonly, free, and uid.
        * lrecord.h (set_lheader_implementation): Adjust to new
        lrecord_header.
        * lrecord.h (struct lrecord_implementation): The field basic_p
        for indication of an old lrecord is not needed anymore, remove.
        * lrecord.h (MAKE_LRECORD_IMPLEMENTATION): Remove basic_p.
        * lrecord.h (MAKE_EXTERNAL_LRECORD_IMPLEMENTATION): Remove
        basic_p.
        * lrecord.h (copy_sized_lrecord): Remove distinction between
        old lrecords and lcrecords.
        * lrecord.h (copy_lrecord): Remove distinction between old
        lrecords and lcrecords.
        * lrecord.h (zero_sized_lrecord): Remove distinction between
        old lrecords and lcrecords.
        * lrecord.h (zero_lrecord): Remove distinction between old
        lrecords and lcrecords.
```

4.7 Remove lcrecords and lcrecord lists

The next step was the allocation of the former lcrecords with the new allocator. The lcrecord lists were only built to keep track of lcrecords within the old allocator. This data structure was the reason for various hacks (e.g. `XD_FLAG_FREE_LISP_OBJECT` during KKCC marking). The lcrecord lists are not needed any more, they are completely removed.

All other functions used to manage old lcrecords are also removed, allocation is done by the new lrecord allocation functions described above. From now on there is only one kind of lisp object allocated on the heap: the lrecord.

The various Lisp objects all over the code allocated as lcrecords are modified: they all get the new `lrecord_header` and are allocated using the new allocator's lrecord functions.

Note: These changes occur in many files, but are quite simple.

`src/ChangeLog addition:`

```
        Remove lcrecords and lcrecord lists:

        * alloc.c (basic_alloc_lcrecord): Not needed anymore, remove.
        * alloc.c (very_old_free_lcrecord): Not needed anymore, remove.
        * alloc.c (copy_lisp_object): No more distinction between
        lrecords and lcrecords.
        * alloc.c (all_lcrecords): Not needed anymore, remove.
        * alloc.c (make_vector_internal): Allocate as lrecord.
        * alloc.c (make_bit_vector_internal): Allocate as lrecord.
        * alloc.c: Completely remove ‘lcrecord lists’.
        * alloc.c (free_description): Remove.
        * alloc.c (lcrecord_list_description): Remove.
        * alloc.c (mark_lcrecord_list): Remove.
        * alloc.c (make_lcrecord_list): Remove.
        * alloc.c (alloc_managed_lcrecord): Remove.
        * alloc.c (free_managed_lcrecord): Remove.
        * alloc.c (alloc_automanaged_lcrecord): Remove.
        * alloc.c (free_lcrecord): Remove.
        * alloc.c (lcrecord_stats): Remove.
        * alloc.c (tick_lcrecord_stats): Remove.
        * alloc.c (disksave_object_finalization_1): Add call to
        mc_finalize_for_disksave. Remove the lcrecord way to visit all
        objects.
        * alloc.c (kkcc_marking): Remove XD_FLAG_FREE_LISP_OBJECT
        * alloc.c (sweep_lcrecords_1): Remove.
        * alloc.c (common_init_alloc_early): Remove everything related
        to lcrecords, remove old lrecord initializations,
        * alloc.c (init_lcrecord_lists): Not needed anymore, remove.
```

```
* alloc.c (reinit_alloc_early): Remove everything related to
lcrecords.
* alloc.c (init_alloc_once_early): Remove everything related to
lcrecords.
* buffer.c (allocate_buffer): Allocate as lrecord.
* buffer.c (nuke_all_buffer_slots): Use lrecord functions.
* buffer.c (common_init_complex_vars_of_buffer): Allocate as
lrecord.
* buffer.h (struct buffer): Add lrecord_header.
* casetab.c (allocate_case_table): Allocate as lrecord.
* casetab.h (struct Lisp_Case_Table): Add lrecord_header.
* charset.h (struct Lisp_Charset): Add lrecord_header.
* chartab.c (fill_char_table): Use lrecord functions.
* chartab.c (Fmake_char_table): Allocate as lrecord.
* chartab.c (make_char_table_entry): Allocate as lrecord.
* chartab.c (copy_char_table_entry): Allocate as lrecord.
* chartab.c (Fcopy_char_table): Allocate as lrecord.
* chartab.c (put_char_table): Use lrecord functions.
* chartab.h (struct Lisp_Char_Table_Entry): Add lrecord_header.
* chartab.h (struct Lisp_Char_Table): Add lrecord_header.
* console-impl.h (struct console): Add lrecord_header.
* console-msw-impl.h (struct Lisp_Devmode): Add lrecord_header.
* console-msw-impl.h (struct mswindows_dialog_id): Add
lrecord_header.
* console.c (allocate_console): Allocate as lrecord.
* console.c (nuke_all_console_slots): Use lrecord functions.
* console.c (common_init_complex_vars_of_console): Allocate as
lrecord.
* data.c (make_weak_list): Allocate as lrecord.
* data.c (make_weak_box): Allocate as lrecord.
* data.c (make_ephemeron): Allocate as lrecord.
* database.c (struct Lisp_Database): Add lrecord_header.
* database.c (allocate_database): Allocate as lrecord.
* device-impl.h (struct device): Add lrecord_header.
* device-msw.c (allocate_devmode): Allocate as lrecord.
* device.c (nuke_all_device_slots): Use lrecord functions.
* device.c (allocate_device): Allocate as lrecord.
* dialog-msw.c (handle_question_dialog_box): Allocate as lrecord.
* elhash.c (struct Lisp_Hash_Table): Add lrecord_header.
* elhash.c (make_general_lisp_hash_table): Allocate as lrecord.
* elhash.c (Fcopy_hash_table): Allocate as lrecord.
* event-stream.c: Lcrecord lists Vcommand_builder_free_list and
Vtimeout_free_list are no longer needed. Remove.
* event-stream.c (allocate_command_builder): Allocate as lrecord.
* event-stream.c (free_command_builder): Use lrecord functions.
* event-stream.c (event_stream_generate_wakeup): Allocate as
lrecord.
```

```
* event-stream.c (event_stream_resignal_wakeup): Use lrecord
functions.
* event-stream.c (event_stream_disable_wakeup): Use lrecord
functions.
* event-stream.c (reinit_vars_of_event_stream): Lcrecord lists
remove.
* events.h (struct Lisp_Timeout): Add lrecord_header.
* events.h (struct command_builder): Add lrecord_header.
* extents-impl.h (struct extent_auxiliary): Add lrecord_header.
* extents-impl.h (struct extent_info): Add lrecord_header.
* extents.c (allocate_extent_auxiliary): Allocate as lrecord.
* extents.c (allocate_extent_info): Allocate as lrecord.
* extents.c (copy_extent): Allocate as lrecord.
* faces.c (allocate_face): Allocate as lrecord.
* faces.h (struct Lisp_Face): Add lrecord_header.
* file-coding.c (allocate_coding_system): Allocate as lrecord.
* file-coding.c (Fcopy_coding_system): Allocate as lrecord.
* file-coding.h (struct Lisp_Coding_System): Add lrecord_header.
* fns.c (Ffillarray): Allocate as lrecord.
* frame-impl.h (struct frame): Add lrecord_header.
* frame.c (nuke_all_frame_slots): Use lrecord functions.
* frame.c (allocate_frame_core): Allocate as lrecord.
* glyphs.c (allocate_image_instance): Allocate as lrecord.
* glyphs.c (Fcolorize_image_instance): Allocate as lrecord.
* glyphs.c (allocate_glyph): Allocate as lrecord.
* glyphs.h (struct Lisp_Image_Instance): Add lrecord_header.
* glyphs.h (struct Lisp_Glyph): Add lrecord_header.
* gui.c (allocate_gui_item): Allocate as lrecord.
* gui.h (struct Lisp_Gui_Item): Add lrecord_header.
* keymap.c (struct Lisp_Keymap): Add lrecord_header.
* keymap.c (make_keymap): Allocate as lrecord.
* lisp.h (struct Lisp_Vector): Add lrecord_header.
* lisp.h (struct Lisp_Bit_Vector): Add lrecord_header.
* lisp.h (struct weak_box): Add lrecord_header.
* lisp.h (struct ephemeron): Add lrecord_header.
* lisp.h (struct weak_list): Add lrecord_header.
* lrecord.h (struct lcrecord_header): Not used, remove.
* lrecord.h (struct free_lcrecord_header): Not used, remove.
* lrecord.h (struct lcrecord_list): Not needed anymore, remove.
* lrecord.h (lcrecord_list): Not needed anymore, remove.
* lrecord.h: (enum data_description_entry_flags): Remove
XD_FLAG_FREE_LISP_OBJECT.
* lstream.c: Lrecord list Vlstream_free_list remove.
* lstream.c (Lstream_new): Allocate as lrecord.
* lstream.c (Lstream_delete): Use lrecod functions.
* lstream.c (reinit_vars_of_lstream): Vlstream_free_list
initialization remove.
```

```
* lstream.h (struct lstream): Add lrecord_header.
* emacs.c (main_1): Remove lstream initialization.
* mule-charset.c (make_charset): Allocate as lrecord.
* objects-impl.h (struct Lisp_Color_Instance): Add
lrecord_header.
* objects-impl.h (struct Lisp_Font_Instance): Add lrecord_header.
* objects.c (Fmake_color_instance): Allocate as lrecord.
* objects.c (Fmake_font_instance): Allocate as lrecord.
* objects.c (reinit_vars_of_objects): Allocate as lrecord.
* opaque.c: Lcreord list Vopaque_ptr_list remove.
* opaque.c (make_opaque): Allocate as lrecord.
* opaque.c (make_opaque_ptr): Allocate as lrecord.
* opaque.c (free_opaque_ptr): Use lrecord functions.
* opaque.c (reinit_opaque_early):
* opaque.c (init_opaque_once_early): Vopaque_ptr_list
initialization remove.
* opaque.h (Lisp_Opaque): Add lrecord_header.
* opaque.h (Lisp_Opaque_Ptr): Add lrecord_header.
* emacs.c (main_1): Remove opaque variable initialization.
* print.c (default_object_printer): Use new lrecord_header.
* print.c (print_internal): Use new lrecord_header.
* print.c (debug_p4): Use new lrecord_header.
* process.c (make_process_internal): Allocate as lrecord.
* procimpl.h (struct Lisp_Process): Add lrecord_header.
* rangetab.c (Fmake_range_table): Allocate as lrecord.
* rangetab.c (Fcopy_range_table): Allocate as lrecord.
* rangetab.h (struct Lisp_Range_Table): Add lrecord_header.
* scrollbar.c (create_scrollbar_instance): Allocate as lrecord.
* scrollbar.h (struct scrollbar_instance): Add lrecord_header.
* specifier.c (make_specifier_internal): Allocate as lrecord.
* specifier.h (struct Lisp_Specifier): Add lrecord_header.
* symbols.c:
* symbols.c (Fmake_variable_buffer_local): Allocate as lrecord.
* symbols.c (Fdontusethis_set_symbol_value_handler): Allocate
as lrecord.
* symbols.c (Fdefvaralias): Allocate as lrecord.
* symeval.h (struct symbol_value_magic): Add lrecord_header.
* toolbar.c (update_toolbar_button): Allocate as lrecord.
* toolbar.h (struct toolbar_button): Add lrecord_header.
* tooltalk.c (struct Lisp_Tooltalk_Message): Add lrecord_header.
* tooltalk.c (make_tooltalk_message): Allocate as lrecord.
* tooltalk.c (struct Lisp_Tooltalk_Pattern): Add lrecord_header.
* tooltalk.c (make_tooltalk_pattern): Allocate as lrecord.
* ui-gtk.c (allocate_ffi_data): Allocate as lrecord.
* ui-gtk.c (allocate_emacs_gtk_object_data): Allocate as lrecord.
* ui-gtk.c (allocate_emacs_gtk_boxed_data): Allocate as lrecord.
* ui-gtk.h (structs): Add lrecord_header.
```

```
        * window-impl.h (struct window): Add lrecord_header.
        * window-impl.h (struct window_mirror): Add lrecord_header.
        * window.c (allocate_window): Allocate as lrecord.
        * window.c (new_window_mirror): Allocate as lrecord.
        * window.c (make_dummy_parent): Allocate as lrecord.

modules/ChangeLog addition:

        Remove Lcrecords:

        * postgresql/postgresql.c (allocate_pgconn): Allocate with new
        allocator.
        * postgresql/postgresql.c (allocate_pgresult): Allocate PGresult
        with new allocator.
        * postgresql/postgresql.h (struct Lisp_PGconn): Add
        lrecord_header.
        * postgresql/postgresql.h (struct Lisp_PGresult): Add
        lrecord_header.
        * ldap/eldap.c (allocate_ldap): Allocate with new allocator.
        * ldap/eldap.h (struct Lisp_LDAP): Add lrecord_header.
```

4.8 MEMORY_USAGE_STATS

For exact calculation of really used memory of the objects I added the function `mc_malloced_storage_size`, which calculates the real size of an object within the heap (including overhead). It is modelled after the already existing `malloced_storage_size`, which considers the overhead of the underlying allocator like `malloc.c`, `gmalloc.c` or system malloc.

The already existing function `show-memory-usage` pretty prints memory usage information about some Lisp objects (buffer, marker, charset, scrollbar, window, and unicode translation tables). For those Lisp objects the `compute_*_usage` functions are needed to calculate the sizes. The calls to `malloced_storage_size` are replaced by `mc_malloced_storage_size`.

`show-memory-usage` also prints the storage usage of all the other Lisp objects. It gets the data for this statistics from `garbage-collect`. The changes needed to collect this information with the new allocator are described in the next section.

The new function `show-mc-alloc-memory-usage` prints information about the different size classes: how big they are, how many objects are allocated in a size class and how much space is left over.

The output looks like this:

```
...
USED HEAP     |       currently in use      |       total available       |
cell-sz #pages| #cells    space    total  % | #cells    space    total  % |  %
-------------------------------------------------------------------------------
      8      1|      8       64       64 100|    256     2048     2048 100|   3
```

```
     12   1176| 115419  1385028  1385028 100| 199920  2399040  2408448  99|  57
     16   1070|  53840   861440   861440 100| 136960  2191360  2191360 100|  39
...
```

The size class 12 contains 1,176 pages right now (`PAGE_SIZE` in this example is 2,048 byte). In size class 12 all objects with a size of 12 byte a kept—in XEmacs this is the storage class for conses (amongst others). So currently 115,491 objects are alife in this size class, in sum they use 1,385,028 byte. The 100% indicate, that the objects are allocated efficiently; the cell size fits exactly the object size, there is no wasted memory within one cell.

The total available space even includes the free cells space. 199,920 cells are available in this size class. In the per page view, memory is wasted: on a page of 2,048 byte fit 170.6 cells with 12 byte. So on every page 8 byte cannot be used. This explains the difference between the total available space (which can be used by objects) and the totally allocated space. Thus, the efficiency only reaches 99%.

The last percentage indicates the filling of this size class: 57% of the cells are currently in use.

These statistics are conditionalized on the (already existing) flag `MEMORY_USAGE_STATS`.

```
lisp/ChangeLog addition:

        MEMORY_USAGE_STATS

        * diagnose.el: Add new lisp function to pretty print statistics
        about the new allocator.
        * diagnose.el (show-mc-alloc-memory-usage): New.

src/ChangeLog addition:

        MEMORY_USAGE_STATS

        * alloc.c (fixed_type_block_overhead): Not used anymore, remove.
        * buffer.c (compute_buffer_usage): Get storage size from new
        allocator.
        * marker.c (compute_buffer_marker_usage): Get storage size from
        new allocator.
        * mule-charset.c (compute_charset_usage): Get storage size from
        new allocator.
        * scrollbar-gtk.c (gtk_compute_scrollbar_instance_usage): Get
        storage size from new allocator.
        * scrollbar-msw.c (mswindows_compute_scrollbar_instance_usage):
        Get storage size from new allocator.
        * scrollbar-x.c (x_compute_scrollbar_instance_usage): Get
        storage size from new allocator.
        * scrollbar.c (compute_scrollbar_instance_usage): Get storage
        size from new allocator.
        * unicode.c (compute_from_unicode_table_size_1): Get storage
        size from new allocator.
        * unicode.c (compute_to_unicode_table_size_1): Get storage size
        from new allocator.
        * window.c (compute_window_mirror_usage): Get storage size from
        new allocator.
        * window.c (compute_window_usage): Get storage size from new
        allocator.
```

4.9 MC_ALLOC_TYPE_STATS

To achieve backwards compatibility, the code reproduces the statistics returned after a garbage collection as closely as possible. This information is used by `show-memory-usage`. It is based on the way the old allocator works: for every single type a separate free list is kept. So it was very easy to get statistics about how many objects of a certain type are currently used and how many free cells for this type are left over (the information about the free cells for a certain type are lost with the new allocator—free lists are not kept for a type, they are kept for a certain size).

With the new allocator, it is easy to get information about how many objects of a certain size are alive. But it is more difficult, to get statistics on a per type basis, since the new allocator does not distinguish between types anymore.

I added the functionality of getting type-based statistics but it consumes a lot of time. This is why it is kept conditionalized on a separate flag called `MC_ALLOC_TYPE_STATS` (which is set by default in `mc-alloc.h`).

The statistics are collected by the allocation and free functions of the new allocator.

For details on collection statistics for the Lisp object string, see Section 4.10 [Strings], page 28.

`src/ChangeLog addition:`

```
        MC_ALLOC_TYPE_STATS:

        * alloc.c (alloc_lrecord): Bump lrecord count.
        * alloc.c (noseeum_alloc_lrecord): Bump lrecord count.
        * alloc.c (struct lrecord_stats): Storage for counts.
        * alloc.c (init_lrecord_stats): Zero statistics.
        * alloc.c (inc_lrecord_stats): Increase the statistic.
        * alloc.c (dec_lrecord_stats): Decrease the statistic.
        * alloc.c (gc_plist_hack): Used to print the information.
        * alloc.c (Fgarbage_collect): Return the collected information.
        * mc-alloc.c (remove_cell): Decrease lrecord count.
        * mc-alloc.h: Set flag MC_ALLOC_TYPE_STATS.
        * emacs.c (main_1): Init lrecord statistics.
        * lrecord.h: Add prototypes for *_lrecord_stats.
```

4.10 Strings

Strings are allocated in two parts, a fix-sized object (containing the length, property list, and a pointer to the actual data) as a lrecord in frob blocks, whereas the actual string data is allocated separately. For short strings, the string data is kept in so called string-chars blocks, which get compactified (and thus relocated) during garbage collection. Big strings are individually mallocd.

For now, the allocation of strings in two parts is still kept, but will change in the future. See Chapter 5 [TODO], page 35 for future plans.

The old allocator kept track about the number of living short and big strings. It used the UNMARK_string macro to update the statistics. The `UNMARK_*` macros were called after the garbage collection to reset the mark bits within the objects to zero. This hack to the unmarking phase is no longer possible with the new allocator (the mark bits can be zeroed more efficiently). It leads to a drawback for strings:

- There are no more statistics about short and big strings available.
- The debug_string_purity functionality was also done in the `UNMARK_string` macro, and is now gone as well.

Again, the current way strings are handeled is just an interim solution: See Chapter 5 [TODO], page 35 for future work on strings.

`src/ChangeLog`

```
        Strings:

        * alloc.c (Fmake_string): Initialize ascii_begin to zero.
        * alloc.c (gc_count_num_short_string_in_use): Remove.
        * alloc.c (gc_count_string_total_size): Remove.
        * alloc.c (gc_count_short_string_total_size): Remove.
        * alloc.c (debug_string_purity): Remove.
        * alloc.c (debug_string_purity_print): Remove.
        * alloc.c (sweep_strings): Remove.
```

4.11 Remove static C-readonly Lisp objects

One of my goals was to have all Lisp objects be treated the same way, they should all be managed by the new allocator. But many Lisp objects are built statically e.g. `Lisp_Subr` and `Lisp_Symbol_Value_Forward`. They were denoted as C-readonly Lisp objects, with the c-readonly bit set in the old lrecord header.

To have them managed by the new allocator, the objects are allocated dynamically and the contents of the static structs are copied to their locations in the heap. They now can be treated as normal Lisp objects without any write restrictions. This also leads to a better locality, the alife objects live in the same memory area.

`src/ChangeLog addition:`

```
        Remove static C-readonly Lisp objects:
```

```
        * alloc.c (c_readonly): Not needed anymore, remove.
        * alloc.c (GC_CHECK_LHEADER_INVARIANTS): Remove some obsolete
        lheader invariants assertions.
        * buffer.c (DEFVAR_BUFFER_LOCAL_1): Allocate dynamically.
        * console.c (DEFVAR_CONSOLE_LOCAL_1): Allocate dynamically.
        * gpmevent.c: Indirection via MC_ALLOC_Freceive_gpm_event.
        * gpmevent.c (Fgpm_enable): Allocate dynamically.
        * gpmevent.c (syms_of_gpmevent): Allocate dynamically.
        * lisp.h (C_READONLY): Not needed anymore, remove.
        * lisp.h (DEFUN): Allocate dynamically.
        * lrecord.h (C_READONLY_RECORD_HEADER_P): Not needed anymore,
        remove.
        * lrecord.h (SET_C_READONLY_RECORD_HEADER): Not needed anymore,
        remove.
        * symbols.c (guts_of_unbound_marker):
        * symeval.h (defsubr): Allocate dynamically.
        * symeval.h (DEFSUBR_MACRO): Allocate dynamically.
        * symeval.h (DEFVAR_ SYMVAL_FWD): Allocate dynamically.
        * tests.c (TESTS_DEFSUBR): Allocate dynamically.
```

4.12 mcpro

In some cases (i.e. removing the c-readonly functionality) the heap pointers have to be added to the root set. The c-readonly property implicitly added these objects to the root set (made them not collectibile). Since this is gone, I added a new way to add heap pointers to the root set, called mcpro.

Currently, there are only two spots, where this actually needed:

- strings made with `make-string-nocopy`
- `Qunbound` in `init_symbols_once_early`

To add a heap pointer `ptr` to the heap set, simply use `mcpro (ptr)`. The pointer is added to the mcpro dynamic array and used as a root of accessibility during garbage collection.

`src/ChangeLog addition:`

```
        Definition of mcpro:

        * lisp.h: Add mcpro prototypes.
        * alloc.c (common_init_alloc_early): Add initialization for
        mcpros.
        * alloc.c (mcpro_description_1): New.
        * alloc.c (mcpro_description): New.
        * alloc.c (mcpros_description_1): New.
        * alloc.c (mcpros_description): New.
        * alloc.c (mcpro_one_name_description_1): New.
        * alloc.c (mcpro_one_name_description): New.
        * alloc.c (mcpro_names_description_1): New.
        * alloc.c (mcpro_names_description): New.
        * alloc.c (mcpros): New.
        * alloc.c (mcpro_names): New.
        * alloc.c (mcpro_1): New.
        * alloc.c (mc_pro): New.
        * alloc.c (garbage_collect_1): Add mcpros to root set.

        Usage of mcpro:

        * alloc.c (make_string_nocopy): Add string to root set.
        * symbols.c (init_symbols_once_early): Add Qunbound to root set.
```

4.13 Changes to the Portable Dumper

After loading the dump image the objects are living in a read-only memory area, with no allocator information given. I changed this behaviour: during loading the dump file the dumped objects are handed one by one to the new allocator. Then they are managed by the new allocator and can be treated as normal objects without any restrictions: they can be overwritten and even be freed during garbage collection.

To achieve this, the storage for the objects is dynamically allocated, the object is copied from the dumped data to the new heap location, and the pointers in the dumped objects are adjusted. This all happens at the time, the dump file is loaded. My changes are slightly slowing down the process of loading the dump file, for the benefit of creating fully privileged objects, efficiently managed by the new allocator.

For the relocation at load time more information about the dumped objects is needed than there is currently provided. I added more information about the objects to the dump file:

- the element count (for array-like data structures).
- the size of the object
- the address of the object's position in the dumped data

This data is currently called 'mc-allocation-table'.

With count and size the right amount of storage can be allocated for the object. The object's position is used to build a lookup table, where the object's position within the dumped data is used to map to the new address in the heap. This lookup table is used for relocation: by using the relocation table from the dump file all pointers are identified and updated with the result of the lookup.

My changes lead to a slightly different structure of the dump file:

```
 0              - header
                - dumped objects
  stab_offset   - mc allocation table (count, size, address) for individual
                  allocation and  relocation at load time.
                - nb_cv_data*struct(dest, adr) for in-object externally
                  represented data
                - nb_cv_ptr*(adr) for pointed-to externally represented data
                - relocation table
                - nb_root_struct_ptrs*struct(void *, adr)
                  for global pointers to structures
                - nb_root_blocks*struct(void *, size, info) for global
                  objects to restore
                - root lisp object address/value couples with the count
                  preceding the list
```

Loading the dumped data takes the following steps:

1. The elements of the mc-allocation-table are loaded one-by-one. For each Lisp object, `mc_alloc` is used to allocate memory; for all other objects, `xmalloc` is used.
2. The object is copied from its position in the dumped data to the new memory location.
3. The position in the dumped data and the new heap address are used to built the lookup table.
4. After all dumped objects are restored, the relocation table is used to identify the heap pointers in the dumped data.
5. With the help of the lookup table all pointers within the objects are updated.
6. The static variables are updated (with the help of the lookup table).

It occurred, that the hash tables of the dumper uses for address translations are not performing very good with the new allocator. Therefore, I increased the size of these hash tables and modified the the hash function, to better suit the new alloctor.

For more information about the dump process in general, see section "Dumping" in *XEmacs Internals Manual*.

`src/Changelog`

```
        Changes to the Portable Dumper:

        * alloc.c (FREE_OR_REALLOC_BEGIN): Since dumped objects can be
        freed with the new allocator, remove assertion for !DUMPEDP.
        * dumper.c: Adjust comments, increase PDUMP_HASHSIZE.
        * dumper.c (pdump_make_hash): Shift address only 2 bytes, to
        avoid collisions.
        * dumper.c (pdump_objects_unmark): No more mark bits within
        the object, remove.
        * dumper.c (mc_addr_elt): New. Element data structure for mc
        hash table.
        * dumper.c (pdump_mc_hash): New hash table: `lookup table'.
        * dumper.c (pdump_get_mc_addr): New. Lookup for hash table.
        * dumper.c (pdump_get_indirect_mc_addr): New. Lookup for
        convertibles.
        * dumper.c (pdump_put_mc_addr): New. Putter for hash table.
        * dumper.c (pdump_dump_mc_data): New. Writes the table for
        relocation at load time to the dump file.
        * dumper.c (pdump_scan_lisp_objects_by_alignment): New.
```

```
Visits all dumped Lisp objects.
* dumper.c (pdump_scan_non_lisp_objects_by_alignment): New.
Visits all other dumped objects.
* dumper.c (pdump_reloc_one_mc): New. Updates all pointers
of an object by using the hash table pdump_mc_hash.
* dumper.c (pdump_reloc_one): Replaced by pdump_reloc_one_mc.
* dumper.c (pdump): Change the structure of the dump file, add
the mc post dump relocation table to dump file.
* dumper.c (pdump_load_finish): Hand all dumped objects to the
new allocator and use the mc post dump relocation table for
relocating the dumped objects at dump file load time, free not
longer used data structures.
* dumper.c (pdump_load): Free the dump file.
* dumper.h: Remove pdump_objects_unmark.
* lrecord.h (DUMPEDP): Dumped objects can be freed, remove.
```

4.14 New configure flag for `DUMP_IN_EXEC`

Since the dumped data is not used directly, each dumped object has to be handed to the allocator. Although this is slowing down the start of a dumped XEmacs, the dumped data can be treated like normal objects, especially they are no longer write protected and can also be collected if they are not used anymore. See Section 4.13 [Changes to the Portable Dumper], page 30 for more details.

The dump image has only to be present in memory while restoring the dumped objects, which is done immediately after XEmacs is launched. After that, the memory of the dump image can be freed.

If the dump image is written into the executable, this memory cannot be freed, and thus the dumped objects are using twice their size in memory (in the dump image and allocated with the new allocator).

Dumping into the executable is really convenient: you just have to take care about one file and you don't have to deal with (e.g. copy into binary directory) ugly `.dmp`-files, but it is wasting some memory.

I think, the decision how to deal with it should be left to the user: if the user is picky about efficient memory usage, he probably wants to have a separate dump file; if the user does not like to have a separate dump file, he can still dump into the executable. Since memory is really cheap today and the dump in executable is quite cool, users may cope with inefficient memory usage.

Therefore I added the configure flag '`--enable-dump-in-exec`' which is defaulted to on without mc-alloc and defaulted to off with mc-alloc (of course this only causes any impact if the portable dumper is used).

Note: this is the only part of my patch, which is not conditionalized on `MC_ALLOC`. If you prefer to have it committed in a separate patch, let me know and I'll do that.

For backwards compatibility, I modified configure.in (for autoconf 2.13) as well es configure.ac (for autoconf 2.59).

```
ChangeLog addition:

        DUMP_IN_EXEC:

        * Makefile.in.in: Condition the installation of a separate dump
        file on !DUMP_ON_EXEC.
        * configure.ac (XE_COMPLEX_ARG_ENABLE): Add
        ‘--enable-dump-in-exec’ as a new configure flag.
        * configure.ac: DUMP_IN_EXEC is define as default for PDUMP but
        not default for MC_ALLOC.
        * configure.in (AC_INIT_PARSE_ARGS): Add ‘--dump-in-exec’ as a
        new configure flag.
        * configure.in: DUMP_IN_EXEC is define as default for PDUMP but
        not default for MC_ALLOC.
        * configure.usage: Add description for ‘dump-in-exec’.

lib-src/ChangeLog addition:

        DUMP_IN_EXEC:

        * Makefile.in.in: Only compile insert-data-in-exec if
        DUMP_IN_EXEC is defined.

src/ChangeLog addition:

        DUMP_IN_EXEC:

        * Makefile.in.in: Linking for and with dump in executable only if
        DUMP_IN_EXEC is defined.
        * config.h.in: Add new flag ‘DUMP_IN_EXEC’
        * emacs.c: Condition dump-data.h on DUMP_IN_EXEC.
        * emacs.c (main_1): Flag ‘-si’ only works if dump image is
        written into executable.
```

4.15 Miscellanious

Here are the left overs. Dunno how to categorize this stuff...

```
src/ChangeLog addition:

        Miscellanious

        * lrecord.h (enum lrecord_type): Added numbers to all types,
        very handy for debugging.
        * xemacs.def.in.in: Add mc-alloc functions to make them visible
        to the modules.
```

5 TODO

0. Update XEmacs Internals Manual.

   Already working on it...

1. Optimize the new allocator.

   The new allocator can be optimized in many ways, by adjusting some important constants to better suit the allocation of objects for XEmacs:

   - Size of one page (`PAGE_SIZE`)
   - Size classes of the used and unused heap.
   - The growth of the heap

   I am going to do more testing on this (even thinking about a test suite to determine optimal values for different systems, in connection with the new garbage collector).

2. Remove the special case allocation for strings.

   Merge the separately allocated data part into the Lisp object.

   This does not sound hard, but a first simple approach is not working, because a string can be resized via 'resize_string'. This leads to a relocation of the string, which leaves the problem how to update all the pointers to this string.

   Another approach, to make the separate allocation somewhat more efficient, was the usage of the 'unamanged heap' of the new allocator (see Section 3.11 [Unmanaged Heap], page 11) for the data part of the string. This would lead to efficiently allocated storage and would us allow to get rid of the time consuming compaction of the string char blocks after every garbage collection. This change should go hand in hand with the allocation of the dumped non-Lisp objects: instead of allocating them with `xmalloc`, they should be allocated on the unmanaged heap. ###TODO###: Think about this some more!

3. Find other objects with separated out parts and merge them.

   Another candidate for this change is a buffer, which is, like a string, allocated in two parts.

4. NEW GARBAGE COLLECTOR

   That is the final goal; with the new allocator a good basis is set.

6 Benchmarks

I ran some benchmarks to document the current state of the work. Please keep in mind that the new allocator is work in progress and that there are currently no optimizations made at all! The new allocator will perform best with a new garbage collector.

In this benchmark, I use different configurations for compiling XEmacs. The results are listed below and allow direct comparison between the old and the new allocator for different configurations.

I was interested in five different features:

`mc-alloc` The new allocator: ‘`--enable-mc-alloc`’

`use-kkcc` The new mark algorithm: ‘`--enable-kkcc`’.

`pdump` The protable dumper: ‘`--enable-pdump`’

`pdump-in-exec`
The dump file is stored in the executable: ‘`--enable-dump-in-exec`’ in connection with ‘`--enable-pdump`’.

`mule` Multi language support: ‘`--enable-mule`’

If a feature of these five is not listed in the first column of the tables below, it is disabled in this configuration (‘`--disable-*`’).

There are two tables below: the first shows some configurations of a non-mule XEmacs, the second one shows those configurations with a mule-enabled XEmacs.

Here is a description of the collected data:

time measurements
Compare the time it takes XEmacs to complete the following tasks (it measures the the fully elapsed time from starting the command till it is completed):

check The time needed to run `make check`:

```
/usr/bin/time make check
```

This gives a general overview about the overall performance of XEmacs.

start The time needed to start XEmacs (i.e. the loading of the dumped data for pdump-enabled XEmacs). Command:

```
/usr/bin/time src/xemacs -vanilla -kill
```

This is especially interesting to compare the dumper modifications for the new allocator.

bench The time needed to run `(bench 1)` from the benchmark package. Measured with the command:

```
/usr/bin/time src/xemacs -vanilla \
  -l /home/crestani/src/xemacs/bench/bench.el \
  -eval "(bench 1)" -kill
```

This also gives a general overview about the overall performance of XEmacs.

memory usage

This data is collected by using the command `top`:

```
src/xemacs -vanilla eval '(debug-print \
  (shell-command-to-string (concat "top -bp " \
  (int-to-string (emacs-pid)) " -n 1")))' -kill
```

VIRT Virtual Image (kb): The total amount of virtual memory used by the task. It includes all code, data and shared libraries plus pages that have been swapped out.

CODE Code size (kb): The amount of physical memory devoted to executalbe code, also known as the 'text resident set' size or TRS.

DATA Data+Stack size (kb): The amount of physical memory devoted to other than executable code, also known as the 'data resident set' size or DRS.

file sizes Especially to take a look at the differences between dump image in a separate file and dump image in the executable.

xemacs The file size (kb) of the XEmacs executable `src/xemacs`.

dump The file size (kb) of the XEmacs dump file `src/xemacs.dmp`.

The benchmarks are run on an up-to-date Linux machine with a 3,0GHz Intel Pentium 4 processor and 512MB memory.

6.1 Results for no-mule XEmacs

```
no mule            time measurements       memory usage [kb]      file sizes [kb]
features          check   start   bench   VIRT    CODE    DATA    xemacs  dump
===============================================================================
<no feature>      0:17.89 0:04.74 0:55.72 10.728  4.504    6.224   8.358    -
mc-alloc          0:20.73 0:04.48 1:02.80 18.232  4.468   13.764   8.699    -
-------------------------------------------------------------------------------
pdump             0:17.27 0:04.67 0:54.37 11.148  1.940    9.208   5.491  2.430
mc-alloc pdump    0:20.47 0:04.59 1:03.14 18.652  1.964   16.688   5.728  3.527
-------------------------------------------------------------------------------
pdump-in-exec     0:17.88 0:05.17 0:54.56 11.128  4.308    6.820  10.041  2.430
mc-alloc
pdump-in-exec     0:21.08 0:04.58 1:03.29 22.208  5.512   16.696  10.279  3.527
-------------------------------------------------------------------------------
kkcc              0:21.54 0:05.34 1:02.36 10.700  4.480    6.220   8.372    -
mc-alloc kkcc     0:23.99 0:04.52 1:06.13 18.248  4.492   13.756   8.709    -
-------------------------------------------------------------------------------
kkcc pdump        0:20.17 0:05.13 0:57.47 11.108  1.904    9.204   5.494  2.430
mc-alloc kkcc
pdump             0:23.30 0:05.76 1:11.00 18.624  1.928   16.696   5.735  3.527
-------------------------------------------------------------------------------
kkcc
pdump-in-exec     0:20.94 0:05.27 0:58.12 11.120  4.296    6.824  10.045  2.430
mc-alloc kkcc
pdump-in-exec     0:23.35 0:04.83 1:07.09 22.144  5.444   16.700  10.286  3.527
```

###TODO###: Figure out, where exactly all this memory (8MB difference) is going.

6.2 Results for mule-enabled XEmacs

```
with mule          time measurements       memory usage [kb]     file sizes [kb]
features          check   start   bench   VIRT    CODE    DATA    xemacs  dump
================================================================================
mule              0:51.70 0:06.73 0:59.65 12.300  5.284    7.016   9.368    -
mc-alloc mule     1:47.69 0:06.25 1:07.50 20.532  5.332   15.200   9.936
--------------------------------------------------------------------------------
mule pdump        0:45.74 0:06.80 0:58.51 12.772  2.092   10.680   5.897  3.121
mc-alloc
mule pdump        1:41.60 0:07.14 1:07.74 21.104  2.116   18.988   6.145  4.540
--------------------------------------------------------------------------------
mule
pdump-in-exec     0:46.53 0:06.50 0:50.72 12.784  5.160    7.624  10.449  3.121
mc-alloc mule
pdump-in-exec     1:42.11 0:06.76 1:08.44 25.632  6.644   18.988  10.696  4.540
--------------------------------------------------------------------------------
mule kkcc         2:49.63 0:07.12 1:12.27 12.336  5.300    7.036   9.378    -
mc-alloc mule
kkcc              3:34.85 0:06.24 1:18.44 20.564  5.352   15.212   9.950    -
--------------------------------------------------------------------------------
mule kkcc
pdump             2:43.18 0:06.95 1:11.51 12.808  2.064   10.744   5.903  3.121
mc-alloc mule
kkcc pdump        3:28.28 0:07.00 1:18.57 21.084  2.104   18.980   6.151  4.540
--------------------------------------------------------------------------------
mule kkcc
pdump-in-exec     2:47.97 0:07.35 1:12.77 12.804  5.132    7.672  10.457  3.121
mc-alloc mule kkcc
pdump-in-exec     3:32.85 0:06.80 1:19.29 25.616  6.632   18.984  10.702  4.540
```

Index

Short Contents

Table of Contents