

# An Extended Low Fat Allocator API and Applications

Gregory J. Duck

Department of Computer Science  
National University of Singapore  
gregory@comp.nus.edu.sg

Roland H. C. Yap

Department of Computer Science  
National University of Singapore  
ryap@comp.nus.edu.sg

## Abstract

The primary function of memory allocators is to allocate and deallocate chunks of memory primarily through the `malloc` API. Many memory allocators also implement other API extensions, such as deriving the size of an allocated object from the object’s pointer, or calculating the base address of an allocation from an interior pointer. In this paper, we propose a general purpose extended allocator API built around these common extensions. We argue that such extended APIs have many applications and demonstrate several use cases, such as (manual) memory error detection, meta data storage, typed pointers and compact data-structures. Because most existing allocators were not designed for the extended API, traditional implementations are expensive or not possible.

Recently, the LowFat allocator for heap and stack objects has been developed. The LowFat allocator is an implementation of the idea of low-fat pointers, where object bounds information (size and base) are encoded into the native machine pointer representation itself. The “killer app” for low-fat pointers is automated bounds check instrumentation for program hardening and bug detection. However, the LowFat allocator can also be used to implement highly optimized version of the extended allocator API, which makes the new applications (listed above) possible. In this paper, we implement and evaluate several applications based efficient memory allocator API extensions using low-fat pointers. We also extend the LowFat allocator to cover global objects for the first time.

## 1 Introduction

Memory allocators are used heavily in languages without garbage collection, for example, in C/C++. Memory allocation (and deallocation), canonically this is through `malloc/free` (or C++’s `new` operators), is well understood and studied [18]. There are many widely used memory allocators, to name a few, the Lea [2], jemalloc [1], and TCMalloc [4]. Most allocators provide APIs for allocating (`malloc` and friends) and deallocation (`free` and friends). For brevity, we will simply call this the *malloc API*.

The nub of the `malloc` API has remained fairly static for a long time, focusing on the core functionality of allocation and deallocation of memory. However, there is other functionality which can be offered, separate from the main

allocation and deallocation tasks. Indeed, some allocators provide some extended non-core functionality, and we argue that extensions, such as returning information about allocated objects, is both useful and can support a variety of applications. While some allocators have some non-core `malloc` API extensions, we propose a unifying set of `malloc` API extensions.

Our extensions leverage the LowFat allocator which has been recently developed for efficient bounds checking [8, 10]. The LowFat allocator allows for certain operations, such as calculating the allocation size/base/offset of pointers very efficiently, which forms the foundation of our API extensions. This is important for applications where the extended API is heavily used, e.g., in bounds checking potentially every read/write can make use of LowFat operations. Our API extension also allows for uniform treatment of all objects (globals, stack and heap), in contrast, traditional memory allocators only provide APIs for heap objects. Although some similar APIs already exist – e.g., the Boehm conservative garbage collector [6] also provides some similar functionality since the garbage collector also needs some of the operations we propose – by exploiting the properties of LowFat pointers, our implementation is very efficient, with many operations implementable in a few inlined low-latency instructions. While low-fat pointers have been implemented for heap [8] and stack [10] objects, in this paper we also extend low-fat pointers to also cover global objects, thereby covering all three main object kinds.

We show how to apply the extended `malloc` API to several applications, including: (manual) memory error checking, efficient and general meta data storage and retrieval, typed pointers, and compact data-structures. For each application we provide some (mini)benchmarks to support our claims. Berger et al. [5] propose the need for composable memory allocators, here, we argue the case for applications which leverage new functionality beyond memory allocation/deallocation.

In summary, the main contributions of this paper are the following:

- *Low-fat Globals*: In addition to heap and stack objects, we extend low-fat pointers to also cover global objects for the first time. This means that low-fat pointers are now applicable to all three main object kinds: heap, stack and globals.

- *An Extended LowFat Allocator API*: We present an extended version of the `malloc` API which gives additional operations outside the core allocation functionality. The extended API leverages low-fat pointers which allows for very efficient implementation of key operations.
- *Applications*: We present several novel applications, made possible by the extended `malloc` API, for non-traditional use cases, including: manual memory error checking; hidden meta-data; typed/tagged pointers; and compact vectors. We also evaluate the applications to show that they are efficient either from a time or space perspective.

The paper is organized as follows: Section 2 summarizes the existing LowFat allocator for heap and stack objects, and then we present a novel extension for low-fat global objects. We also evaluate the performance of the LowFat allocator against some more established competitors. Section 3 presents the LowFat allocator extended API, as well as details the efficient implementation of each operation. Finally, in Section 4, we present and evaluate several applications of the extended LowFat allocator API.

## 2 LowFat Allocation Design and Implementation

This section describes the LowFat allocator’s design and implementation. In a memory allocator, the precise system details can be important. Throughout this paper, we will tailor the implementation details for the `x86_64` architecture and Linux operating system.

### 2.1 Background: Low-fat Pointers

Low-fat pointers [8, 10, 14] are a method for encoding object *bounds information* (object’s size and base) into the native machine pointer representation itself. For example, a highly simplified low-fat pointer encoding may be implemented as follows:

```
union { void *ptr;
    struct {uintptr_t size:10; // MSB
           uintptr_t unused:54; } meta;} p;
```

Here the object size is represented explicitly as a field `size`, and the base address can be encoded implicitly by ensuring object’s are aligned to an address that is a multiple of `size`, thus  $base(p) = p - (p \bmod p.size)$ . Crucially we see that a low-fat pointer is the same size as a machine pointer, i.e.  $(sizeof(p) == sizeof(void*))$ . Low-fat pointers generally require a machine architecture with sufficient bit-width, i.e., 48 or 64bit pointers, such as the `x86_64`.

This simplified low-fat pointer encoding is difficult to implement in practice as it imposes strong constraints on the program’s virtual address space layout. Instead we focus on the *flexible* low-fat pointer encoding of [8, 10], which we shall refer to as *LowFat*. The general idea of LowFat is to

partition the program’s virtual address space into several large equally-sized *regions*. There are two main types of regions: *low-fat regions* which contain objects managed by the LowFat allocator, and *non-fat regions* that contain everything else. In [8], region #0 is non-fat, and we will also follow that approach. The basic idea is that each low-fat region will service allocations of a given size range, as illustrated in Figure 1. For example, region #1 handle allocations of sizes 1-16 bytes, region #2 handles sizes 17-32 bytes, region #3 33-48 bytes, etc. The mapping between sizes and low-fat regions is called the *size configuration* [8], represented by a sequence *Sizes*. For example, the *Sizes* for [10] is as follows:

$$Sizes = \langle 16, 32, 48, 64, 80, 96, 112, 128, 144, \dots \rangle$$

Generally, the size configuration should have the following properties:

1. All sizes must be a multiple of 16bytes;
2. *Sizes* must include a power-of-two sub-sequence, i.e.:  $Sizes \cup \langle 16, 32, 64, 128, 256, \dots \rangle = Sizes$ ; and
3. Large multi-page sizes should be powers-of-two, i.e.:  $Sizes \cup \langle 16KB, 32KB, 64KB, \dots \rangle = Sizes$ ; and

Property 1 ensures the allocator obeys the default alignment of standard `malloc` for 64-bit systems. Property 2 is needed to support both the stack and global low-fat pointers (discussed below) as well as support for the `mema1ign` API. Property 3 keeps  $|Sizes|$  compact, since large multi-page objects can be “rounded-up” to the nearest power-of-two multiple without wasting memory (the “padding” will remain virtual). Note that properties 1, 2 and 3 are consistent with each other. The full low-fat allocator parameters used in this paper are listed in Appendix A.

During allocation, an object of *size* is rounded-up to the next allocation size ( $allocSize \geq size$ ) that fits, which some caveats discussed below. For the object *O* to qualify as *low-fat*, two main properties must be satisfied:

- **Region**: The object *O* is allocated from the sub-heap in region #*I*, where  $Sizes[I] = allocSz$ ; and
- **Alignment**: The object *O* is  $allocSz$ -aligned.

These two properties ensure that the object’s size and base address can be quickly calculated from a (possibly interior) pointer to the object *O*. This will be elaborated on in Section 3.

Memory for the low-fat regions is created during program initialization, e.g., as a `preinit_array` callback. Regions do not grow or shrink during program execution, rather, the initial size is assumed to be large enough to accommodate all “reasonable” future memory requirements of the program. For example, the implementation of [10] assumes a region size of 32GB. The low-fat regions are initially *virtual memory* reserved using `mmap` using the `NORESERVE` flag, and thus does not initially consume any RAM/swap resources. Memory resources are only consumed for the parts of each region that are actually allocated and used by the program. Finally,

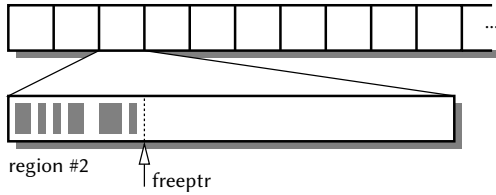


Figure 1. LowFat memory layout.

each region is further partitioned into three heap/stack/global *sub-regions* to handle allocations of the corresponding memory type (see Figure 1). This will be discussed further below.

## 2.2 LowFat Heap Allocation

The exact memory allocation algorithm used for heap objects within each region is left open. The [8, 10] implementation uses a simple free-list allocator design that partitions the heap sub-region into *used* and *unused* space. Objects in the *used* space are either allocated and in use by the program, or have been freed and placed on a “free list” awaiting re-allocation. When a call to `lowfat_malloc(s)` occurs, the LowFat allocator:

1. Determines which region  $#i$  corresponding to size  $s$  should the allocation be serviced from; and
2. Pops an entry from the free-list for region  $#i$  if non-empty; else
3. Allocate a new object from the *unused* space otherwise.

Calls to `free(p)` are handled by pushing the allocated space pointed to by  $p$  onto the corresponding free-list. For large objects, it is sometimes necessary to return free’ed memory back to the operating system, which is done using the `madvise` system call with the `DONTNEED` flag.

Since all allocations of a particular size class are serviced from a single region, this has the side-effect of simplifying the overall allocator design. For example, merging of adjacent free objects is disallowed, thus the corresponding logic to do so is not needed by the allocator. The trade-off is that this may lead to more fragmented memory since free’ed objects can only be reallocated as objects within the same size class. On the other hand, since the allocation size can be determined from the pointer (i.e., which region does the pointer point to?), and since there is no need to implement adjacent free object merging, the LowFat allocator also eliminates the need to store an explicit “malloc header”, meaning that objects are tightly packed. In contrast, the standard `stdlib malloc` implementation for Linux appends a 16byte header to every object. That said, we highlight that in this paper, the main aim of the LowFat allocator is to support an enriched LowFat allocator API presented in Section 3, rather than to design an allocator that directly competes with the current state-of-the-art on performance.

## Benchmarking the LowFat Heap Allocator

We present some benchmarks to evaluate the performance of the LowFat allocator against some more established alternatives. All experiments (including in later sections) are run on a Xeon E5-2630v4 processor (clocked at 2.20GHz) with 32GB of RAM on Linux. The compiler used is LLVM 4.0.0 at `-O2`, and we evaluate against the SPEC2006 benchmark suite. We compare the LowFat implementation of [3] against `stdlib malloc`, `jemalloc` [1], and the Boehm `malloc` (in manual memory management mode) [6]. The results on the SPEC2006 benchmark suite are shown in Figure 2. The geometric mean for `stdlib malloc` is 277.8 (100%), LowFat is 280.9 (101.1%), `jemalloc` is 266.8 (96.0%), and Boehm is 283.6 (102.1%).

Overall we see that the LowFat allocator is competitive against the alternatives. The LowFat allocator described in this paper is intended to be a basic prototype without the many optimizations used in mature memory allocators, so we expect higher overheads compared to more optimized memory allocators such as `jemalloc`. Furthermore, the LowFat allocator is a relatively young system, meaning that further optimizations may be implemented in the future. We also highlight that only the LowFat allocator supports the optimized LowFat API, which is the main focus of this paper. The memory overhead for the LowFat allocator is  $\sim 3\%$  compared to `stdlib malloc` [8, 10].

## 2.3 LowFat Stack Allocation

A LowFat allocator for stack memory is presented in [10], which we briefly summarize here. The low-fat stack allocator works by maintaining a linear mapping between the stack sub-regions (Figure 1) and the main program stack. When the program requests a stack allocation of *size*, the LowFat stack allocator performs the following steps:

1. Round-up *size* to the nearest power-of-two allocation size (*allocSize*) that fits;
2. Mask the stack pointer `%rsp` with  $allocSize - 1$ . This *allocSize*-aligns `%rsp`;
3. Decrement `%rsp` by *allocSize*, allocating space;
4. Map `%rsp` to a pointer *ptr* to the stack sub-region corresponding to *allocSize* using the linear mapping. The *ptr* now points to the newly allocated low-fat stack object.

This mapping is implemented as a compiler transformation [3]. Power-of-two sizes are used since this simplifies object alignment at the cost of increased space overheads. Stack deallocation is handled the same as before, i.e., by restoring `%rsp` to some previous value. The LowFat stack allocation method is similar to the notion of *parallel shadow stacks* [7], but with multiple shadow stacks (one for each sub-region) and some additional steps *allocSize*-aligning objects. Having multiple shadow stacks may waste memory, however, this can be mitigated by mapping each shadow

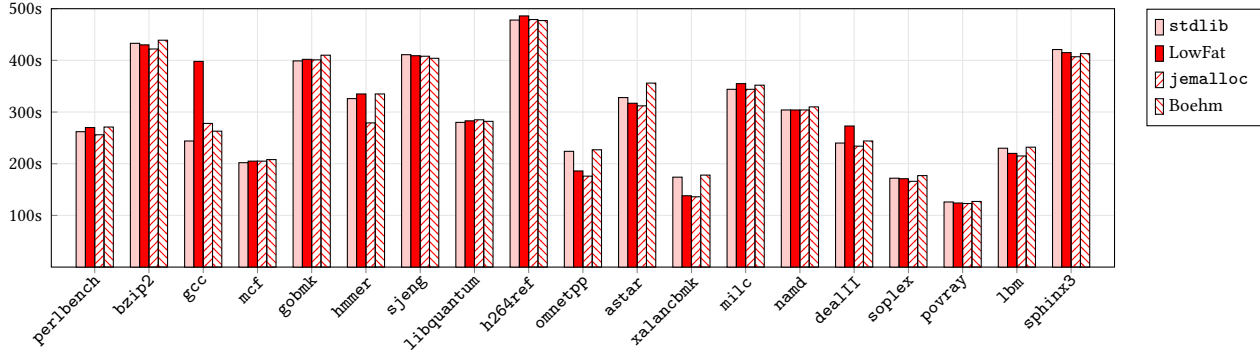


Figure 2. Evaluation of the LowFat and other heap allocators against the SPEC2006 benchmark suite.

stack to the same physical memory. See the *memory aliasing* optimization from [10].

### 2.4 LowFat Global Allocation

Previous work on low-fat pointers are restricted to heap [8] and stack [10] objects only. In this paper, we present an extension of LowFat to also cover global objects. The basic idea is to statically allocate global objects from the global sub-region for the corresponding allocation size. To achieve this, we use a program transformation which annotates global objects using a `section` attribute and then uses a special linker script to control the location of objects. Namely, objects of given *size* are annotated with a

```
attribute(section("lowfat_region_idx"))
```

`section` attribute, where *idx* corresponds to the region index for the global object’s *size*. The static location of the objects can then be controlled via an appropriate *linker script* (ld), e.g.:

```
. = (global sub-region #1 address)
lowfat_region_1 :
    KEEP(*(lowfat_region_1))
...
```

In addition to location, alignment of global objects is controlled using the `aligned` attribute. Due to the power-of-two limitation of the `aligned` attribute, global objects are placed into the nearest power-of-two sized region that fits (as is the case with stack objects).

There are some (compiler tool-chain) caveats for generating global low-fat pointers. Firstly, the *dynamic linker* does not support the `section` directive meaning that dynamically linked globals (e.g., from shared objects) will not be low-fat pointers. This does not affect program behavior but limits the applicability of the LowFat API for such objects. The second caveat is that the compiler may assume all global objects occupy the first 4GB of the virtual address space. This allows the compiler to generate slightly faster code for the x86\_64 architecture. This assumption is violated by global low-fat pointers, meaning that the program must be compiled using

	Operation	Inlined?	Related?
I	lowfat_malloc	✗	✓
	lowfat_realloc	✗	✓
	lowfat_free	✗	✓
	...	...	...
II	lowfat_is_ptr	✓	N.A.
	lowfat_is_heap_ptr	✓	✗
	lowfat_is_stack_ptr	✓	✗
	lowfat_is_global_ptr	✓	✗
III	lowfat_index	✓	N.A.
	lowfat_size	✓	✓*
	lowfat_base	✓	Boehm†
	lowfat_offset	✓	Boehm†
	lowfat_usable_size	✓	Boehm†

Figure 3. Summary of the LowFat API. Here (*Inlined?*) indicates whether the operation can be inlined, and (*Related?*) indicates whether a operation is implemented by some other related malloc API. The caveat (\*) means implemented with the limitation that the pointer must be a base pointer, and (†) means operation is not implemented directly, but can be implemented using the API with minimal effort.

the (`-mcmode=large`) option which disables the assumption. The final caveat is that, like the LowFat stack allocator, global objects are not low-fat by default unless the compiler transformations described in this section are employed.

## 3 LowFat Allocator API

The core motivation for the allocator design is to support the LowFat memory API, as summarized in Figure 3. It is divided into three classes. Class I refers to the (traditional) `malloc` API. The focus of this paper will be on classes II and III detailed below.

### 3.1 Standard allocator functionality

Our LowFat allocator supports standard replacements for `libc`’s memory allocation functions (Figure 3 class I), such as,

`malloc`, `free`, `realloc`, `memalign`, etc. The LowFat replacements are also aliased to versions prefixed by “`lowfat_`”, e.g. `lowfat_malloc`, etc.

Stack and global objects can be transformed automatically as a compiler pass (e.g., as used by [10]). As such, stack and global support is optional, and programmers may opt not to use it.

### 3.2 Core LowFat functionality

The motivation behind LowFat allocation is that allows for some key pointer operations to be implemented efficiently, namely, calculating the size, base, offset, etc., of a pointer `p` with respect to the original allocation. We highlight that the operations take only a few machine instructions making them suitable for inlining which helps efficiency and compiler optimizations. Since these operations are not traditionally supported by the `malloc` API, we refer to these operations as the extended memory allocation API.

By design, unlike the `malloc` API, these operations work uniformly, regardless of whether the pointer is for a heap, stack, global, interior or exterior, just as long as the pointer is lowfat as per Section 2. In Section 4, we will describe some applications of the API.

Given the memory layout of Figure 1, we can define a fundamental operation, `lowfat_index`, that maps a pointer `ptr` to the *region index* to which `ptr` belongs, as follows:

$$\text{lowfat\_index}(ptr) = ptr / \text{LOWFAT\_REGION\_SIZE}$$

Here `LOWFAT_REGION_SIZE` is the region size and is assumed to be a power-of-two. For example, our reference implementation assumes `LOWFAT_REGION_SIZE` is 32GB. Crucially, the `lowfat_index` is fast, compiling down into a single x86\_64 shift instruction with this default:

```
shrq $35,%rax /* 2^35 = 32GB */
```

#### Size (`lowfat_size`)

One common memory allocator API operation is to determine the size of the allocation based on a pointer to an object. This exists in the form of `malloc_usable_size` for `stdlib`’s `malloc`, `HeapSize` for the Window’s `HeapAlloc`, and `GC_size` for the Boehm collector, amongst others. Note that all of these functions assume a pointer to the base of the allocated object. Furthermore, such extensions typically differ on whether the size returned accounts for any additional bytes of padding that may have been added by the allocator. For example, `malloc_usable_size` returns the size including the padding, whereas `HeapSize` returns the original requested allocation size, depending on the version of Windows.

We define `lowfat_size` to return the allocation size of a pointer including any padding, similar to `malloc_usable_size`:

```
lowfat_size(ptr) = LOWFAT_SIZES[lowfat_index(ptr)]
```

Here, `LOWFAT_SIZES` is a constant lookup table mapping region indices to the allocation sizes according to the *size configuration* defined in Section 2. For region indices  $i$  that are *not* associated with LowFat allocation, we define:

$$\text{LOWFAT\_SIZES}[i] = \text{SIZE\_MAX}$$

This definition simplifies some applications relating to bounds checking.

Note that, unlike related allocators, the `lowfat_size` works for any interior pointer and does not assume the base address. The other advantage is that the `lowfat_size` compiles down into two x86\_64 instructions, one shift for `lowfat_index` followed by a memory read:

```
movq LOWFAT_SIZES(,%rax,8),%rbx
```

#### Base (`lowfat_base`) and offset (`lowfat_offset`)

Given a pointer `ptr` to an allocated object  $O$  of size, then

$$\{ptr + 1, \dots, ptr + size\}$$

are the *interior pointers* of  $O$ , and `ptr` is the *base pointer* (a.k.a. exterior pointer) of  $O$ . We can map any (possibly interior) pointer  $ptr' \in I$  to object  $O$  to the base pointer `ptr` using the following operation:

$$\text{lowfat\_base}(ptr) = (ptr / \text{lowfat\_size}(ptr)) * \text{lowfat\_size}(ptr)$$

This assumes 64bit integer arithmetic, and is also equivalent to  $ptr - ptr \% \text{lowfat\_size}(ptr)$ . This also relies on the LowFat allocator ensuring that all allocated objects are *size-aligned*. Assuming the pointer is stored in register `%rax` (and is an implicit argument), and the allocation size in `%rbx`, then the `lowfat_base` operation reduces to two instructions:

```
divq %rbx
imulq %rbx
```

As noted in [8], the 64bit `divq` operation is relatively slow (high throughput and latency [13]), which may not be desirable. There are two main approaches to optimizing `lowfat_base`, namely:

1. Use a power-of-two-only size configuration; or
2. Use fixed-point or floating-point arithmetic.

The first allows for the slow division to be replaced by a fast bitmask operation, for example:

$$\text{lowfat\_base}(ptr) = ptr \& \text{LOWFAT\_MASKS}[\text{lowfat\_index}(ptr)]$$

where `LOWFAT_MASKS[i]` is defined to be  $(\text{LOWFAT\_SIZES}[i] - 1)$  for low-fat region  $\#i$ , or 0 otherwise. The main disadvantage with this approach is that object sizes are rounded to the nearest power-of-two, which leads to increased space overheads. An alternative approach is to use *fixed-point arithmetic* by defining:

$$\text{LOWFAT\_MAGICS}[i] = ((1 \ll R) / \text{LOWFAT\_SIZES}[i]) + 1$$

for low-fat region  $\#i$ , or 0 otherwise. The (+1) term is for error control, see [8] Section 5.1.1. Here  $R$  defines the position of

```

1 void memcpy(void *dst, void *src, int n)
2 {
3     void *dst_base = lowfat_base(dst);
4     size_t dst_size = lowfat_size(dst);
5     void *src_base = lowfat_base(src);
6     size_t src_size = lowfat_size(src);
7     for (int i = 0; i < n; i++) {
8         void *dst_tmp = dst + i;
9         void *src_tmp = src + i;
10        if (isOOB(dst_tmp, dst_base, dst_size))
11            error();
12        if (isOOB(src_tmp, src_base, src_size))
13            error();
14        *dst_tmp = *src_tmp;
15    }
16 }

```

(a) Automatically instrumented (see [10]).

```

1 void memcpy(void *dst, void *src, int n)
2 {
3     size_t dst_size = lowfat_usable_size(dst);
4     if (n > dst_size)
5         error();
6     size_t src_size = lowfat_usable_size(src);
7     if (n > src_size)
8         error();
9     for (int i = 0; i < n; i++)
10        dst[i] = src[i];
11 }

```

(b) Optimally instrumented.

**Figure 4.** Two bounds-check instrumented variants of (simple) memcpy. The instrumentation is highlighted.

the radix point. This approach allows for a more efficient implementation of the base operation that effectively turns a slow division operation into a fast(er) multiplication:

```

lowfat_base(ptr) =
    (((ptr * LOWFAT_MAGICS[lowfat_index(ptr)]) >> R)
     * lowfat_size(ptr))

```

A good value for  $R$  is 64, as this takes advantage of the x86\_64’s 128bit integer multiplier, meaning the  $R$ -right shift operation “compiles away” into a mere register renaming. It is also possible to use floating-point arithmetic, which is more intuitive, by defining:

```

LOWFAT_MAGICS[i] = (1.0 / LOWFAT_SIZES[i])

```

However, fixed-point avoid conversions to-and-from floating point numbers so is generally more efficient. The main disadvantage of fixed/floating point arithmetic is that calculations may be affected by precision errors, which mainly affect large allocations. Using the (+1) term for error control, precision errors will only affect pointers to “near the end” of these large allocations. This problem is mitigated by modifying the allocator to take precision errors into account [8]. Namely, if an object of a given *size* is potentially affected by a precision errors in region #*i*, then the allocator will instead service the allocation from the next (larger) region #( *i* + 1). The maximum possible precision error for each region is calculated in advance [8].

The Boehm conservative garbage collector [6] also supports GC\_base (equivalent to our lowfat\_base) as an  $O(1)$  operation. However, the Boehm implementation is slower and larger (in terms of code size) to that of low-fat pointers. Due to the larger code size, the Boehm GC\_base operation generally cannot be inlined.

Finally, we define a lowfat\_offset operation that returns the difference from the current pointer and the base:

```

lowfat_offset(ptr) = ptr - lowfat_base(ptr)

```

It is also possible to implement the lowfat\_offset directly with fixed-point arithmetic, i.e., by multiplying the fixed-point mantissa by the allocation size. However, since the mantissa represents the least significant bits, a fixed-point implementation of lowfat\_offset is impractical due to precision errors.

### Usable size (lowfat\_usable\_size)

Recall that the lowfat\_size returns the allocation size for the base or any interior pointer to the object, and this size is the same regardless of the pointer’s offset. For many applications, we wish to know how many bytes are left until we reach the end of the allocated space. For this we define:

```

lowfat_usable_size(ptr) =
    lowfat_size(ptr) - lowfat_offset(ptr)

```

For example, given a pointer  $p$  into a buffer *buf*, then the lowfat\_usable\_size operation can determine how many bytes are left inside *buf* from  $p$  until a buffer overflow occurs.

### Tests (lowfat\_is\_ptr, ..., lowfat\_is\_global\_ptr)

It is sometimes useful to test whether a pointer is low-fat or not. The motivation is to allow inter-operation with non low-fat pointers, possibly, from other memory allocators. It can also be useful to test whether or not the pointer is a low-fat heap/stack/global pointer. These operations reduce to simple range tests, e.g.:

```

lowfat_is_ptr(ptr) =
    (ptr >= &(region #1)) && (ptr < &(region #M+1))

```

Here 1.. $M$  ( $M$  is the last region) are the indices of the low-fat regions. The test starts from region #1 as region #0 is non-fat as per [8]. The narrower heap/stack/global variants additionally test which sub-region (see Figure 1) the pointer points to.

## 4 Applications

The LowFat allocator implementation supports efficient implementations of some operations. This enables some applications that would otherwise be too slow for other memory management systems. In this section we explore examples of such applications, including: manual bounds checking, hidden meta-data, typed pointers and compact data-structure representations.

### 4.1 Detecting Memory Errors

Automated bounds check instrumentation is the “killer app” for low-fat pointers, and this idea has been explored by previous literature [8, 10]. The basic idea is to instrument all pointer arithmetic and memory access with an explicit bounds check (`is00B`) defined as follows:

```
(p < base) || (p > base+size-sizeof(*p))  (is00B)
```

Automatic bounds instrumentation follows the schema introduced in [8]. The basic idea is as follows: for all *input* pointers  $q$  (function arguments, return values, or pointer values read from memory), we calculate the bounds *meta information* by calling the `lowfat_size/lowfat_base` operations. For example:

```
void f(int *q) {
    void *q_base = lowfat_base(q);
    size_t q_size = lowfat_size(q); ...
```

Next, for all pointers  $p$  *derived* from an input pointer  $q$  through pointer arithmetic ( $p = q+k$ ) or field access ( $p = \&q \rightarrow \text{field}$ ), we instrument any access to  $p$  with an (`is00B`) check. For example:

```
int *p = q + k;
if (is00B(p, q_base, q_size)) error();
x = *p;      or      *p = x;
```

Such bounds-check instrumentation is implemented as a LLVM [16] compiler pass, see [3].

An automatically instrumented version of a (simple) implementation of `memcpy` is shown in Figure 4a (based off [10] Figure 2). Here the instrumented lines are highlighted, including the bounds meta data calculation using `lowfat_size/lowfat_base` shown in lines 3–6. Automated bounds checking has an overhead of 64% for heap/stack/global objects [3], although lower overheads are possible depending on what optimizations are enabled (generally trading error coverage for speed).

### Manual Bounds Checking

Automatic bounds instrumentation has the advantage in that it requires minimal intervention on behalf of the programmer (e.g., changing the compiler’s flags). However, the automatically generated instrumentation is generally sub-optimal. For example, in the code from Figure 4a, there are two instrumented bounds checks for each iteration of the

loop (one for the read and one for the write). A more “natural”/optimal approach is to check the bounds once for each pointer outside of the loop, as shown in Figure 4b. Here we use `lowfat_usable_size` to determine the number of bytes available in the `src` and `dst` buffers, and verify that this is consistent with the parameter `n`. Such instrumentation can be added manually by the programmer, assuming that objects are allocated using the LowFat allocator.

In principle, the automatic instrumentation could be further optimized, e.g., by using program analysis to automatically transform Figure 4a into 4b. However, program analysis generally has limitations, and cannot optimize all cases. Furthermore, in some applications the programmer needs fine grained control over what to instrument, in order to achieve an acceptable overhead versus security ratio. Thus, the programmer can restrict instrumentation to specific operations (e.g., `memcpy`) or specific pointers to sensitive data.

The overheads of manual bounds checking depend on how much is instrumented.

### Bonus: Finding free API errors

The LowFat API can be also be used to find some memory errors relating to `free`. For example, a stack or global object should not be free’ed:

```
if (lowfat_is_heap_ptr(ptr)) lowfat_free(ptr);
else error();
```

In a similar vein, a pointer which is not the base of a heap object, e.g. an interior heap pointer, should not be free’ed:

```
if (lowfat_is_heap_ptr(ptr) &&
    !lowfat_offset(ptr)) lowfat_free(ptr);
else error();
```

We remark that general use-after-free checking is beyond the scope of the LowFat API. Testing if a pointer is free or not is known to suffer from races (test versus usage) in multi-threaded environments.

### 4.2 Conservative Garbage Collection

Another application of the LowFat allocator is for marking in conservative garbage collection for C/C++. Under this idea, the LowFat heap allocator itself is modified to automatically invoke a mark-sweep collection phase eliminating the need to manually free objects. As is the standard approach, the “mark” phase scans for all objects *reachable* from some *root set* of pointers, typically global and stack memory. Any reachable object is “marked” using internal meta-data associated with each object. Next, a “sweep” frees all unmarked (unreachable) objects since these are no longer referenced by the program. The garbage collector is *conservative* meaning that it does not rely on C/C++ type information — rather any bit pattern that could be a pointer is assumed to be a pointer. The trade-offs for conservative collection are well known, e.g., see [6].

```

1 void mark(void *ptr)
2 {
3     void *base = lowfat_base(ptr);
4     if (base == NULL)
5         return; // Not low-fat
6     if (set_mark(base))
7         return; // Already marked.
8     void **itr = (void **)base,
9         **end = (void **)(base + lowfat_size(base));
10    for (; itr < end; itr++)
11        mark(*itr);
12 }

```

Figure 5. LowFat API enhanced marking algorithm.

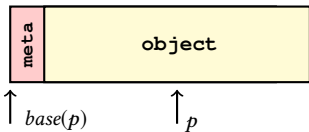


Figure 6. (Hidden) meta-data stored at the base of an object.

The low-fat API can assist with marking algorithm as shown in Figure 5. Here, given a potential pointer value `ptr`, we first check if `ptr` points to a heap object (lines 3–4). Since `ptr` may be an *interior pointer*, i.e., point *inside* an allocated object, we next retrieve the object’s base address by a call to `lowfat_base` (line 5). We assume that `set_mark` marks the object in some disjoint meta-data (lines 6–7), and returns true if the object was already marked (to terminate loops). Finally, we scan the object (lines 8–12) and mark any bitpattern that happens to be a valid pointer. The disjoint meta-data itself is implemented as a collection of bitmaps (one for each region, created using `mmap`), with one bit for every object within the corresponding region.

Note that the Boehm conservative garbage collector [6] implements a similar marking algorithm, but with its own implementations of the size and base operations. This is also one reason why the extended Boehm GC API is similar to the LowFat API.

### 4.3 Hidden Meta-Data

The LowFat API can also be used to associate arbitrary meta-data to allocated objects. The basic idea is to store the meta-data at the base of the object, as illustrated in Figure 6. Here `p` is a (possibly interior) pointer to a LowFat allocated (object), and the meta-data (`meta`) is stored at the base of the allocation. The meta-data can be transparently bound to an object by wrapping memory allocation, such as the following:

```

void *meta_malloc(size_t size, META m) {
    META *ptr = lowfat_malloc(size + sizeof(META));
    *ptr = m;
    return (ptr + 1);
}

```

Note the function returns `(ptr + 1)`, meaning that the meta-data is hidden from the program, analogous to a hidden

`malloc` header that occupies the memory immediately before the allocated object. However, a crucial difference with `malloc` headers is that in `malloc` accessing the header is restricted through a base pointer, here, we have no restrictions. Later, the meta-data can be retrieved via a call to `lowfat_base`, as follows:

```
m = *(META *)lowfat_base(p);
```

The same basic idea can be extended to both stack and global objects, but requires a compiler transformation. Stack allocation is transformed in a similar way to `malloc`, where

```
ptr = alloca(size);
```

is transformed into:

```
META *mptr = lowfat_alloca(size + sizeof(META));
*mptr = m;
ptr = (mptr + 1);
```

Here, `lowfat_alloca` is itself expanded via program transformation, as per [10]. We note that the usage of `alloca` is just for the sake of an example, and the transform is applicable to all forms of stack allocation. In particular, the use of `alloca` can be internal to the compiler as is the case with LLVM.

Globals are more difficult to transform, since a global is also a symbol that may be referenced externally, possibly by code not subject to the automatic program transformation. Thus, we cannot rely on solutions that change the *Application Binary Interface* (ABI). To fix this, we use a simple *symbol-within-a-symbol* trick. The basic idea is as follows: given the original global variable definition:

```
T global = definition;
```

We first define a *wrapper type* of the form:

```
struct wrapper { META m; T data; };
```

We also ensure that the structure is *packed* (e.g. by using the GCC `packed` attribute), meaning that there will be no gap between the `m` and `data` fields. Next, we replace the original global with the wrapped version

```
struct wrapper wrappedGlobal = {m, definition};
```

The program (including external modules) may still reference the original *global* symbol. To fix this we define *global* to point to the `data` field inside *wrappedGlobal*. The most direct way to do this is via (module-level) inline assembly:

```
asm (".globl global"
     ".set global, wrappedGlobal+size");
```

where `size=sizeof(META)`. By using this symbol-within-a-symbol trick, the global variable (*global*) can be used as normal by the program, including by external untransformed modules.

A form of the hidden meta-data approach is used by EffectiveSan [9] to store object dynamic type information, a.k.a., the *effective* type of allocated objects, in order to support dynamic type checking for C/C++. However, there is no limit on the kinds of meta-data that can be stored. Like other generic



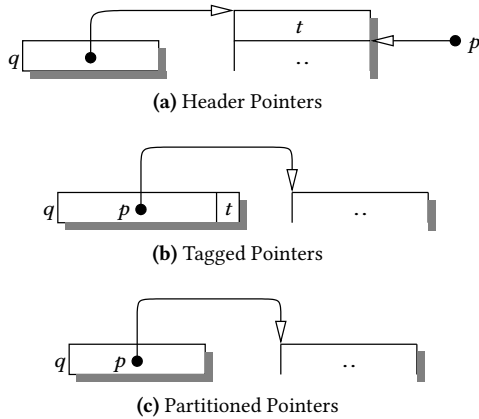


Figure 7. Common typed-pointer representations.

meta-data storage schemes, such as *Padding Area MetaData* (PAMD), there exist many other potential applications, including accurate (exact object size) bounds-checking, profiling and statistics, flow tracking, and data race detection [15]. METAlloc [12] is another general meta data framework, but uses its own shadow memory scheme, and is quite different to our approach and PAMD.

#### 4.4 Typed Pointers

A *typed pointer* is one of various methods for associating dynamic type information with pointers. There are several existing methods [11] for associating a type  $t$  to a pointer  $p$  to form a typed-pointer  $q$ . These include:

- *Headers*: store  $t$  within the object pointed to by  $p$  (Figure 7a);
- *Tagged*: fold  $t$  into the representation of  $p$  itself (Figure 7b);
- *Partitioned*: allocate  $p$  from different regions based on  $t$  (Figure 7c).

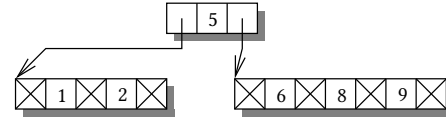
Each approach has its own advantages/disadvantages: *header pointers* is portable but consumes memory to store  $t$ ; *tagged pointers* and *partitioned pointers* do not consume more memory, but rely on knowledge about the underlying memory management system.

In this section we explore some alternatives/extensions based on the LowFat API, namely: *size-typed* pointers and *extended tagged* pointers.

#### Size-typed pointers

One idea is to distinguish pointer types based on the allocation size, a.k.a. *size-typed pointers*. The size can be determined very quickly via the `lowfat_index` API call, however, this approach is only applicable to objects where each supported dynamic type happens to correspond to a different allocation size. That said, real-world applications exist, as illustrated by the following example:

**Example 1 (2-3-4 Trees).** To illustrate size-typed pointers we consider an implementation of *2-3-4 trees* [17]. A 2-3-4 tree is a self-balancing tree data-structure that can be used to implement *associative arrays* mapping keys to values. For example, the following



is a 2-3-4 tree consisting of a root 2-node, a left child 3-node, and a right child 4-node. The name “2-3-4” represents the three node types: *2-nodes*, *3-nodes*, and *4-nodes*, which are of sizes (in 8byte words) of 3, 5, and 7 respectively. This means the nodes will be allocated from different region #2, #3, and #5 respectively, assuming the standard size configuration. Thus, given a pointer `ptr` to a (undetermined) 2-3-4 node, we can efficiently determine the dynamic type by using the `lowfat_index` operation. □

Size-typed pointers are essentially a special case of partitioned pointers. The main advantage is that the LowFat allocator supports the functionality directly, rather than requiring the programmer to implement a specialized allocator.

#### Extended Tagged Pointers

Sized-typed pointers have limited applicability, since the mapping from types to allocation sizes must be one-to-one. Tagged pointers are more general, but the number of tag bits can be limited. For this, we introduce the notion of *extended tagged pointers* which are a generalization of standard tagged pointers using the unused lower  $N$ -bits (typically  $N=4$ ) of allocated objects. Assuming  $N=4$  this allows for 16 distinct types, whereas extended tagged pointers can store up to *size* distinct types, where *size* is the allocation size of the object. Normally, for standard tagged pointers, the type (*tag*) can be retrieved via a simple bitmask operation, e.g.,

$$tag = ptr \& 0xF$$

However, using the LowFat API, we can generalize this as follows:

$$tag = lowfat\_offset(ptr)$$

This supports all possible tag values within the range  $[0..size)$ . Alternatively, tagged pointers may use the unused high bits (typically 16 bits for `x86_64`). Extended tagged pointers may replace or be used in conjunction with high tag bits, depending on the application.

The `lowfat_offset` operation is generally slower than the constant bitmask operation required for standard tagged pointers, especially if fixed-point arithmetic is used. Thus, there is a trade-off between performance and number of types, meaning the usefulness is application dependent. We provide one such application in Section 4.5.

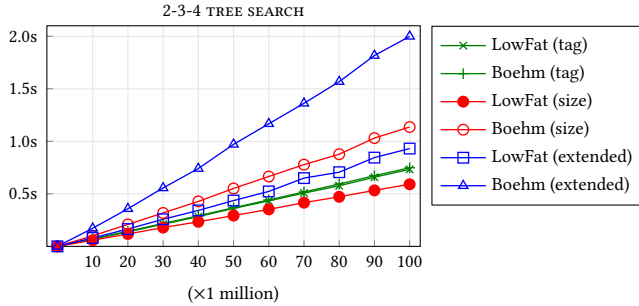


Figure 8. 2-3-4 tree typed pointer performance results.

**Evaluation: 2-3-4 trees**

We evaluate both size-typed and extended tagged pointers for 2-3-4 trees. Our benchmark consists of a searching for every key in a 2-3-4 tree of size  $N$ , measured in seconds. We compare six different versions: a standard tagged pointer implementation (`tag`) using the lower 4 tag bits, an implementation using size-typed pointers (`size`), and an implementation using extended tagged pointers (`extended`). Although extended tagged pointers are overkill for 2-3-4 trees, it is nevertheless a useful test for performance evaluation. We compare each version implemented either the LowFat API, or using the similar Boehm GC API. For the Boehm tests, we use manual memory management mode.

The results are shown in Figure 8. Unsurprisingly, the (`tag`) tests (which do not use any special API calls) show little difference in performance between the two versions. For LowFat, size-typed pointers (`size`) are even faster than traditional tagged pointers by  $\sim 20\%$ . This shows that size-typed pointers are a good alternative for performance critical code, under the caveat that size-typing is applicable to target data-structure. Extended tagged pointers (`extended`) are slower than traditional tagged pointers by  $\sim 27\%$ , so should only be used for applications that require extra tag bits. Also unsurprisingly, the Boehm variants of size-typed and extended tagged pointers were much slower than the LowFat version, e.g.  $> \times 2$  for extended tagged pointers. This is because the LowFat API is highly optimized and inlined for the size/base operations, whereas the Boehm API requires library calls.

**4.5 Low-fat Vectors**

A very common data-structure is a *vector*, for example C++’s `std::vector`, which typically consists of three core components: an array of items `data` (vector data), a length `len` (vector length), and a current position `pos` (next free item). Vectors are normally implemented as structures containing these three components:

```
struct vector {size_t len;
              size_t pos;
              item *data;}
```

We refer to such representations as “fat” vectors.

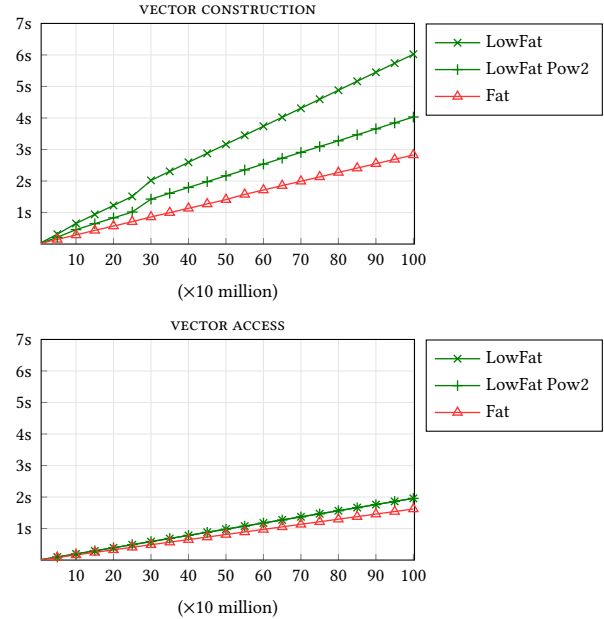


Figure 9. Low-fat vector benchmarks in seconds.

Using the LowFat API we can implement a more compact representation, a.k.a. “low-fat” vectors. For this, we define a vector to be an array of items: (`typedef item *vector`). The `len` field becomes implicit, and can be calculated dynamically using `lowfat_size`:

$$len = lowfat\_size(vector) / sizeof(item)$$

The `pos` can be stored as an *extended tag*, i.e.

```
pos = lowfat_offset(vector)
data = lowfat_base(vector)
```

**Evaluation: Low-fat vectors**

The main advantage of low-fat vectors is that they eliminate the need to explicitly store the `len`, `pos` and `data` fields. Assuming that `len`, `pos`, (`item *`) and `item` are all 1-word in size, then if a fat vector consumes  $n$  words, the corresponding low-fat vector will consume  $(n - 3)$  words. The trade-off is that (re)calculating fields incurs additional overheads compared to storing the values directly. To evaluate the performance of low-fat vectors, we benchmark constructing a single vector of integers using the `push_back` operation. Next, we evaluate the time taken to calculate the sum of all elements of the vector. The results are shown in Figure 9 illustrating the classic space-time tradeoff. We see that constructing low-fat vectors is  $\sim 2\times$  overhead for non-power-of-two sizes,  $\sim 1.33\times$  overhead for power-of-two sizes. Reading from low-fat vector incurs a  $\sim 1.2\times$  overhead for both versions. Thus, low-fat vectors are best suited for programs that create large numbers of small vectors and where optimizing memory overheads are the priority.

## 5 Conclusions

In this paper we presented an extended LowFat memory allocation API. The main advantage of the LowFat API extensions is that some operations, namely, finding the size/base/offset of pointers, relative to the original allocation, are very fast operations (typically can be implemented in a few in-lined instructions). We argue that these properties enable several applications for the LowFat allocator that are not feasible with existing allocators, such as bounds checking, generic meta-data storage, typed pointers and compact data-structures. We evaluated several of these ideas, with promising results. The `malloc` API has been essentially unchanged for a long time, we believe that the idea of memory allocation API extensions going beyond the core allocator function is a genuinely useful and practical addition.

## Acknowledgements

This research was partially supported by a grant from the National Research Foundation, Prime Minister’s Office, Singapore under its National Cybersecurity R&D Program (TSU-NAMi project, No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

## References

- [1] 2018. jemalloc memory allocator. <http://jemalloc.net/>
- [2] 2018. Lea Memory Allocator. <http://g.oswego.edu/dl/html/malloc.html>
- [3] 2018. LowFat: Lean C/C++ Bounds Checking with Low-Fat Pointers. <https://github.com/GJDuck/LowFat>
- [4] 2018. TCMalloc: Thread-Caching Malloc. <http://goog-perftools.sourceforge.net/doc/tcmalloc.html>
- [5] E. Berger, B. Zorn, and K. McKinley. 2001. Composing High-performance Memory Allocators. In *Programming Language Design and Implementation*. ACM.
- [6] H. Boehm and M. Weiser. 1988. Garbage collection in an uncooperative environment. *Software Prac. Experience* 18, 9 (Sept. 1988), 807–820.
- [7] T. Dang, P. Maniatis, and D. Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *ACM Symposium on Information, Computer and Communications Security*. ACM.
- [8] G. Duck and R. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Compiler Construction*. ACM.
- [9] G. Duck and R. Yap. 2018. EffectiveSan: Type and Memory Error Detection using Dynamically Typed C/C++. In *Programming Language Design and Implementation*. ACM.
- [10] G. Duck, R. Yap, and L. Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *Network and Distributed System Security Symposium*. The Internet Society.
- [11] D. Gudeman. 1993. Representing Type Information in Dynamically Typed Languages. Technical Report.
- [12] I. Haller, E. Kouwe, C. Giuffrida, and H. Bos. 2016. METAlloc: Efficient and Comprehensive Metadata Management for Software Security Hardening. In *European Workshop on System Security*. ACM.
- [13] Intel Corporation. 2018. Intel 64 and IA-32 Architectures Optimization Reference Manual.
- [14] A. Kwon, U. Dhawan, J. Smith, T. Knight, and A. DeHon. 2013. Low-fat Pointers: Compact Encoding and Efficient Gate-level Implementation of Fat Pointers for Spatial Safety and Capability-based Security. In *Computer and Communications Security*. ACM.
- [15] Z. Liu and J. Criswell. 2017. Flexible and Efficient Memory Object Metadata. In *International Symposium on Memory Management*. ACM.
- [16] LLVM 2018. <http://llvm.org>.
- [17] R. Sedgewick and K. Wayne. 2011. *Algorithms* (4th ed.). Addison-Wesley Professional.
- [18] P. Wilson, M. Johnstone, M. Neely, and D. Boles. 1995. *Dynamic Storage Allocation: a Survey and Critical Review*. Springer.

## A Low-fat Parameters

LOWFAT\_REGION\_SIZE = 32GB

M = |Sizes| = 61

(16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 192, 224, 256,  
272, 320, 384, 448, 512, 528, 640, 768, 896, 1024, 1040,  
1280, 1536, 1792, 2048, 2064, 2560, 3072, 3584, 4096,

Sizes = 4112, 5120, 6144, 7168, 8192, 8208, 10240, 12288,  
16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB,  
2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB,  
256MB, 512MB, 1GB, 2GB, 4GB, 8GB)