# DFWMalloc High-performance Replacement malloc Library

Paul Sheer http://dfwmalloc.us

Aug 2021



## Overview

The goal of this development is to produce a malloc library that consumes substantially less clock ticks per malloc/free than any existing malloc library, for heavyweight enterprise server software, by exploiting the CPU memory cache efficiently, while at the same time being highly resistant to fragmentation. This means DFWMalloc performs well when CPU cache pressure is high for long-running programs.

DFWMalloc is a drop-in replacement for existing C Library malloc and C++ new/delete operators, preserving compatibility for well-written programs. DFWMalloc can be preloaded for all executables on a typical Linux/FreeBSD installation by placing it in the “preload” path. See **Installation**.

The library works using the Unix mmap() system call and does not make use of brk()/sbrk().

DFWMalloc is highly configurable at compile-time. All the constants and sizes mentioned below are compile-time configurable to remove any arbitrary limitations for your unique project, and allow customization of behavior.

**Thus it is possible to trade speed against memory-efficiency simply by choosing different compile-time options.**

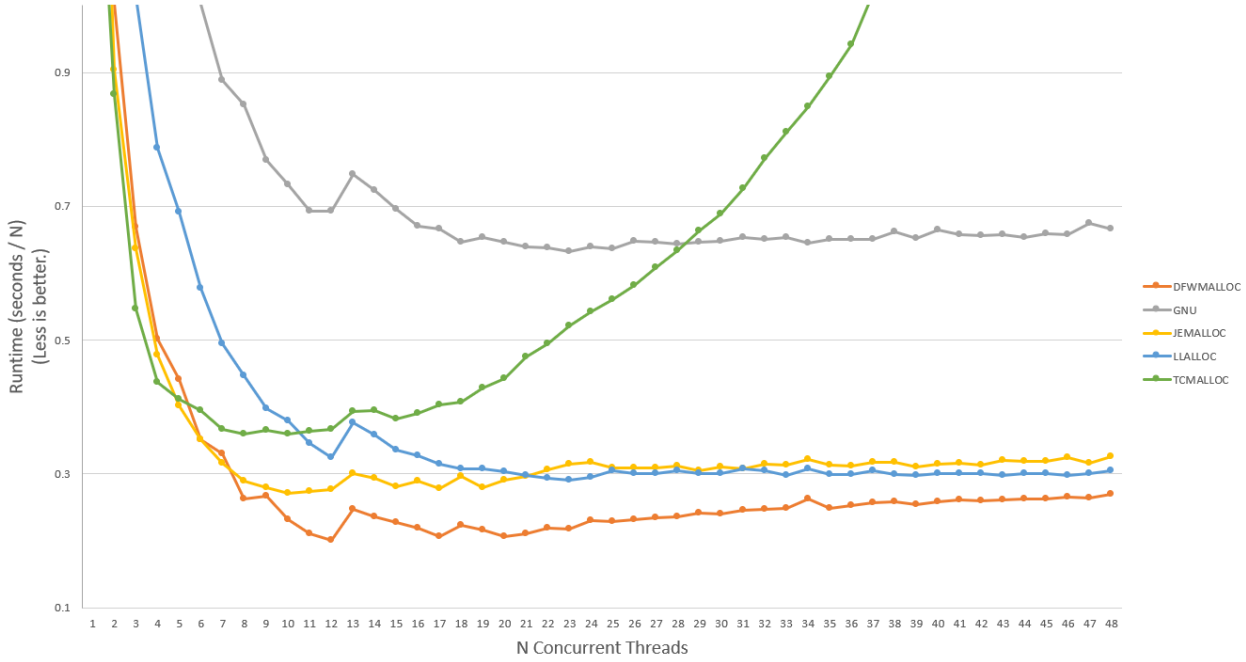
DFWMalloc supports a range of tracing and accounting features for debugging C/C++ malloc usage errors, leaks, and out-of-bound writes. For security-critical applications, a “crypto” mode is supported that randomizes the position of blocks.

# Performance

*1. CPU Performance*

DFWMalloc performs well against well-known malloc libraries over a variety of test conditions. There are no test scenarios found where DFWMalloc exhibits extreme-outlier behavior. An example test run, using up to 5GBytes of physical RAM, is graphed in Figure 1. The performance of DFWMalloc is 15% to 30% faster than the next best malloc library for most load scenarios. This test run uses a modified version of *t-test1.c*; see **Appendix A** for source code differences and further discussion:

Figure 1: t-test1.c run performance, with command-line options
N N 20000000 10000 40000, 24-core 4,000 bogomips




Legacy malloc libraries were not included in the comparison since their performance was especially poor.

*2. Code Size*

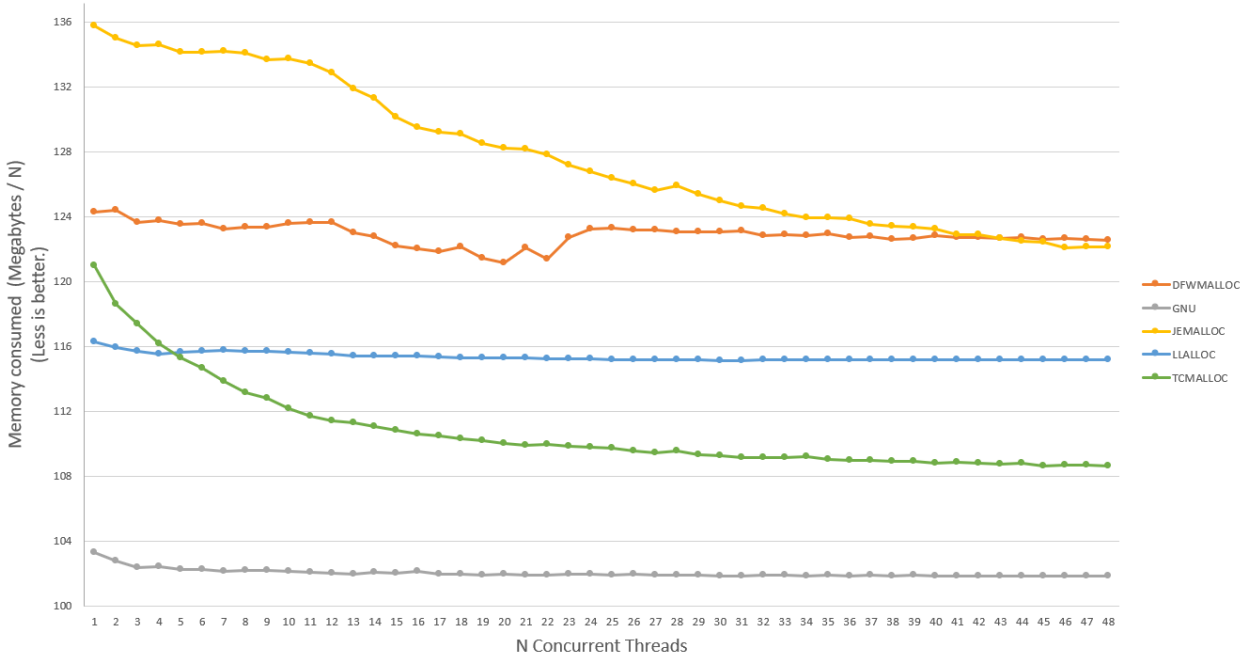
The object code size for each tested malloc library is as follows:

| kB | Library |
|---|---|
| 171 | DFWMALLOC |
| ?? | GNU |
| 575 | JEMALLOC |
| 36 | LLALLOC |
| 154 | TCMALLOC |

*3. Memory Performance*

For the present test scenario DFWMalloc memory efficiency is mostly better than JEMalloc as shown in Figure 2. Memory performance of DFWMalloc can be up to 20% worse than JEMalloc for *worst-case* test scenarios, although such test scenarios do not represent likely real-world loads. See **Appendix A** for discussion.

Figure 2: t-test1.c memory performance, with command-line options
N N 20000000 10000 40000, 24-core 4,000 bogomips



## Installation

*Note that certain applications try to implement their own memory allocators or memory mappings that are incompatible with DFWMalloc. These may crash making your system unbootable should you install DFWMalloc system-wide. Currently* systemd*,* php*, and* ibus *are packages known to not work with DFWMalloc.*

DFWMalloc builds with the command,

```
make
```

To perform a system-wide installation of DFWMalloc on Linux do as follows, copy libdfwmalloc.so to a standard directory for libraries, such as /lib64/.

```
sudo cp libdfwmalloc.so /lib64/
```

Next, create or edit the file /etc/ld.so.preload. The contents of /etc/ld.so.preload are a colon separated list. Add libdfwmalloc.so to the list or, if ld.so.preload is empty, then just add a single line:

```
/lib64/libdfwmalloc.so
```

To use libdfwmalloc.so for a single executable, run as follows:

```
LD_PRELOAD=/lib64/libdfwmalloc.so ./myexe
```

Do not attempt to use LD_PRELOAD at the same time as /etc/ld.so.preload: this will crash your program.

The default build also creates *libdfwmalloc-compact.so* and *libdfwmalloc-enterprise.so*. The *libdfwmalloc-compact.so* version has more memory-efficient build-configuration options for smaller programs. The *libdfwmalloc-enteprise.so* library is more efficient for larger programs that tend to use 100s of megabytes or gigabytes of RAM.

The automated test suite can be optionally run with,

```
make test
```

The very simplest way of including DFWMalloc into your program is simply to add the file dfwmalloc.c to your source tree. Use the compile options: -DUSE_TLS -DHAVE_PTHREAD as follows:

```
gcc -O3 -DUSE_TLS -DHAVE_PTHREAD -c -o dfwmalloc.o dfwmalloc.o
```

Be sure to link with -ldl -lpthread

## Pointer Structure

DFWMalloc library works by storing information about the allocation in the pointer bits itself. Thus pointers returned from malloc() are both a memory offset as well as a bit-encoded message. The bits have the meaning as in Figure 3.

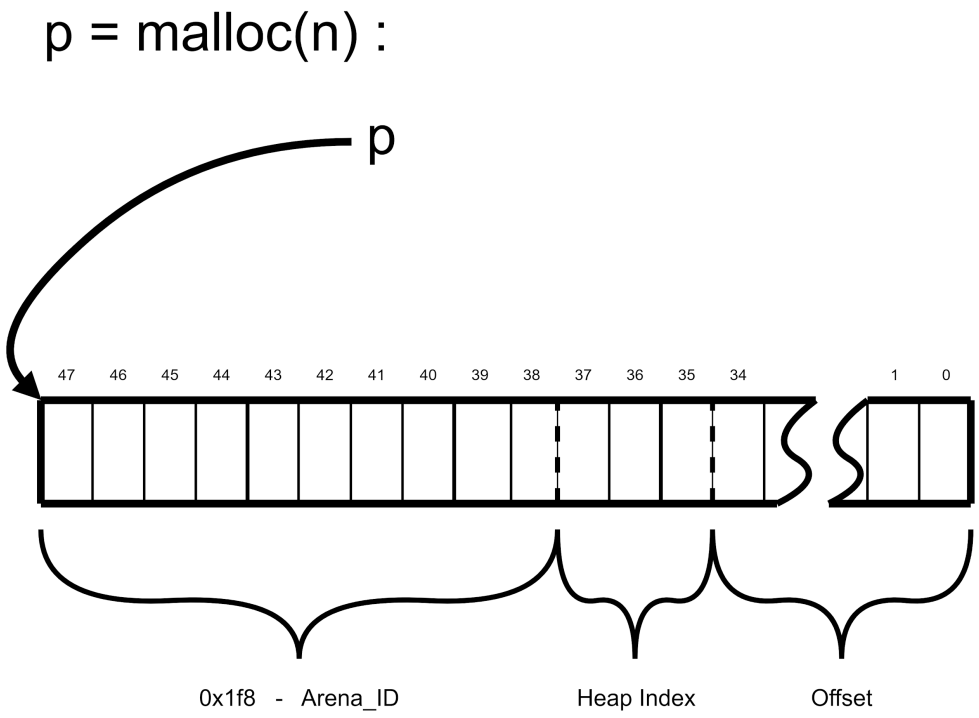



Figure 3: *A pointer returned by malloc.*

Allocations can be of two types: Regular allocations and Control Block (CB) allocations. Regular allocations are allocations more than 65520 bytes. CB allocations are allocations 65520 and less.

## Arenas

DFWMalloc is designed with multithreading in mind. Each execution thread of your program gets its own entirely separate mmap'd region for memory allocation called an *arena*. DFWMalloc divides the 48-bit Intel address space into 1024 arenas. Each active arena receives a unique *arena ID*. The arena ID is encoded into the pointer returned from calls to malloc, realloc, etc. as shown in Figure 3.

For most memory operations arenas are accessed by DFWMalloc *without* any locking.

If you free a pointer that was malloc'd in a different thread, then the free'ing thread will detect that the arena does not match its own. It will not immediately free the block but instead place the block onto a pending list. When the original thread does another malloc or free it will complete the free operation on all blocks on the pending list. This happens transparently to the user: i.e. the programmer may blindly use malloc() and free() just as per POSIX specifications. The only proviso is that free'ing a block from a different thread than it was malloc'd should be done cautiously since there is a performance penalty.

The special case where a thread has a single sibling free'ing thread, is implemented by a *lock-free queue*. This means one free'ing thread can be the free'er for many allocating threads, but a thread must have no more than one free'ing thread. There is a performance penalty if this model is violated, the cost being as much as a 5-fold, since DFWMalloc will be forced internally to acquire locks.

Arenas avoid memory regions reserved by the operating system. For example, the region around 0x5500 0000 0000 is avoided since this is where Linux "ASLR" works.

In order to avoid additional regions of address space, you can use the compile options AVOID_ADDRESS_RANGE_n_START and AVOID_ADDRESS_RANGE_n_END where n is 1 through 5. For example, to avoid the region 0x4000 0000 0000 through 0x47FF FFFF FFFF, use the compile options -DAVOID_ADDRESS_RANGE_1_START=0x400000000000 -DAVOID_ADDRESS_RANGE_1_END=0x480000000000 . Note that these options will reduce your total number of available regions: you may want to increase N_ARENAS_BITS in this case.

## Regular Allocations

A regular allocation uses an exponentially increasing free list that allows allocations up to any size that can fit into *Offset* bits in Figure 3, i.e., approximate $2^{35}$ bytes. The allocation mechanism is minimalist. The following behavior is supported:

1. If a free() of the last block occurs, then the mmap'd portion of the tail end of the heap is shortened, thus freeing memory back to the operating system.
2. If a realloc() occurs of the last block, and if the realloc() enlarges the block, then the mmap'd region is extended to obviate having to "memmove" the block. Thus successive realloc's that enlarge a block are efficient. The converse behavior for a realloc() of the last block shortening the block also applies.
3. The C Library call calloc() always uses a regular allocation and avoids memset() for freshly mmap'd blocks.
4. If back-tracing support is enabled (see **Compile Options**) then all malloc's and free's are watermarked with a call-trace using the backtrace() C Library call. All malloc's, free's,

and double-free's are tracked for error detection. The programmer can thus rely on perfect double-free detection and nonrepudiation.

5. If back-tracing support is compiled-in then all blocks are fenced with magic numbers to detect overflow errors.
6. Back-tracing support must be both compiled-in and also enabled, to track all mallocs and free's. See **Back-tracing and Verification Support**.
7. Back-tracing is supported for regular allocations only. If back-tracing support is enabled, then CB allocations are disabled regardless of the block size.

## CB Allocations

CB allocations pack many memory blocks into one "control block" having an elegant size. This is sometimes called "slab" allocation in other allocators. The default build utilizes 6 possible sizes: 1024, 4096, 16384, 65536, 262144, and 1048576. A control block has the conceptual structure as follows:

```
struct control_block_X {
      /* various management pointers to locate and sort the block: */
      struct control_block *p1;
      ...
      /* various management flags and counts: */
      int f1;
      int c1;
      ...
      /* list of free pointers: */
      char *free_list[N_BLOCKS];
      /* allocated blocks */
      char block[BLOCK_SIZE * N_BLOCKS];
      char pad[WAIST];
};
```

There are in the order of 100 combinations of N_BLOCKS and BLOCK_SIZE for BLOCK_SIZE ranging from 1 to 65520 bytes. However *sizeof(struct control_block_X)* always comes out as one of the six sizes mentioned. DFWMalloc has pre-calculated *const* internal tables with all this information. Each combination is called a *head index*.

Each Arena is further divided into 8 regions labelled with an index 0 through 7. The index is called the *heap index* and is encoded into all pointers returned from malloc. (See Figure 3.)

Each head index has its own free list. The structure of separating free lists and heap size categories gives DFWMalloc its fragmentation resistance. Empty control blocks of the same size may move between heads, and empty control blocks are stored in a *rank-pairing heap* so that the free block with the lowest address number is chosen.

The heap indices 0 through 5 reference the 6 possible sizes of control block mentioned, such that a control block of a particular size gets allocated only from within its distinct region. Heap indices 6 and 7 are for regular allocations.

Notice that allocations are stored adjacent to one another whereas management data is stored at the start of the control block: This has favorable implications for security.

The practical implementation of `char *free_list[N_BLOCKS];` is as one of three kinds of list sets: As a heap-ordered vector; as a bitwise inclusive-or mask of free blocks; and as an unordered vector. The bitwise set consumes only 1 bit overhead per block. In general DFWMalloc attempts to put small blocks contiguously in memory to take advantage of cache lines. These kinds of list sets are known as the **Packing Styles** of a Control Block.

## Packing Styles

Control Blocks need to pack their malloc() blocks adjacent to each other and record a list of free'd blocks. There are several ways to do this.

- An ordinary linked list of free'd blocks is referred to as L packing.
- A ranked heap of pointers to free'd blocks is referred to as H packing. A ranked heap has the advantage that a malloc() of a previously free'd block will choose the lowest ordered block and thus tends to pack blocks contiguously. This has cache-line performance advantages if the blocks are small.
- A bitmask list stores 1 bit for each free'd block. In theory this is algorithmically inefficient, however in practice, using 64-bit bitwise arithmetic and small sets, this is faster than H packing while having the same advantages. This is referred to as S packing.
- The last method is the same as S packing except that the storage size is not a multiple of two and thus the offset cannot be computed by a simple bitwise shift. This is called M packing.
- The blocks are crypto-randomly ordered for reasons of obfuscation, to mitigate against heap attacks on insecure code. This is called C packing. See **Crypto Mode**.

The choice of H packing is currently bypassed for the reason explained. The compile option -DCOMPACT_MODE tends to prefer bitmask packing over list packing.

## Tuning

Many tunable build parameters allow for DFWMalloc to be optimized for a particular environment. See **Compile Options**. The default parameters were selected based on a range

of experiments, however each real-world application will have specific requirements. For infrastructures that run the same application on a large number of virtual OS instances, it makes sense not to confine usage to the default parameters.

For example, large infrastructures that wish to gain a small improvement in memory usage or speed can experiment on a small percentage of nodes. Each iteration of this experiment is economical.

As an example, consider the following contrived compile settings,

```
-DUSE_TLS
-DHAVE_PTHREAD
-DCOMPACT_MODE
-DALLOCATION_SIZE_RATIO=1666
-DCHUNK_POWER=1
-DCB_SIZE_BITS=4
-DBLOCKS_FORMULA=(10000000 / (n + 2000))
```

These produce a range of Control Blocks (See **CB Allocations**) as follows:

| number of blocks | heap index | type | offset | n |
|---|---|---|---|---|
| 4025 | 5 | 1 | 560 | 8 |
| 4059 | 6 | 1 | 576 | 16 |
| 4077 | 7 | 1 | 576 | 32 |
| 4086 | 8 | 2 | 576 | 64 |
| 4675 | 9 | 2 | 656 | 112 |
| 4091 | 9 | 1 | 576 | 128 |
| 2338 | 9 | 2 | 352 | 224 |
| 4093 | 10 | 1 | 576 | 256 |
| 2426 | 10 | 2 | 368 | 432 |
| 2047 | 10 | 1 | 320 | 512 |
| 1213 | 10 | 2 | 208 | 864 |
| 1023 | 10 | 1 | 192 | 1024 |
| 612 | 10 | 2 | 144 | 1712 |
| 511 | 10 | 1 | 128 | 2048 |
| 306 | 10 | 2 | 96 | 3424 |
| 255 | 10 | 1 | 96 | 4096 |
| 153 | 10 | 2 | 80 | 6848 |
| 127 | 10 | 2 | 80 | 8240 |
| 76 | 10 | 2 | 80 | 13792 |
| 63 | 10 | 2 | 64 | 16640 |
| 37 | 10 | 2 | 64 | 28336 |
| 31 | 10 | 2 | 64 | 33808 |
| 18 | 10 | 2 | 64 | 58240 |
| 16 | 10 | 2 | 64 | 65520 |

The row starting with “511” has the meaning that 511 mallocs are stacked into a single control block, starting at 128 bytes offset into the control block, with 2048 bytes per malloc, and administered using bitmasks of type “1”. The total size of the control block is “10” meaning 1 megabyte.

All this information can be obtained from the file *staticdata.c* after running *make*.

## Case Study

*1. Legacy application*

A particular high-performance C++ software application used a range of internal allocation mechanisms. These were all removed and refactored to use standard malloc(), free(), new, and delete. DFWMalloc was then linked with the application and functional verification was performed. Performance testing was then done using the software’s existing performance test suite. After many rounds of tuning, the following optimum settings were established according to the performance requirements:

```
BLOCKS_FORMULA=(5000000 / (n + 2000))
USE_TLS
HAVE_PTHREAD
NO_ASSERT_CHECKS
COMPACT_MODE
ALLOCATION_SIZE_RATIO=1020
CB_SIZE_BITS=4
```

The results were compared to JEMalloc: With the aforementioned compiled settings, DFWMalloc used 20%-25% less memory than JEMalloc while maintaining the same speed.

The build system was then modified to build separate “debug” and “performance” versions so that developers could easily enable the call-tracing feature of DFWMalloc when required.

*2. Small processes*

A system takes advantage of processes that have very small heap sizes. To use small, fast control blocks, the following settings were chosen:

```
BLOCKS_FORMULA=64
USE_TLS
HAVE_PTHREAD
NO_ASSERT_CHECKS
```

```
NO_BLOCK_ORDERING
ALLOCATION_SIZE_RATIO=1200
```

These settings ensure that malloc/frees use an unordered list, which is the fastest for a process that operates entirely in cache. Most control blocks are packed with approximately 64 mallocs.

Testing these settings reveal that DFWMalloc can malloc and free a pointer in 30 CPU clock cycles (sum for both calls).

*3. Benchmark optimization*

A performance-critical environment was able to characterize their malloc usage using a performance benchmark tool which took a few seconds to run. A script was written to cycle through every combination of setting, in order to find settings that minimized run time. The variables were ESTIMATED_MIN_BLOCKS, COMPACT_MODE, BLOCKS_FORMULA, CHUNK_POWER, ALLOCATION_SIZE_RATIO, CB_SIZE_BITS, NO_BLOCK_ORDERING, and MIN_ALIGN. In all, over 20,000 variations of parameters were tested. Within the result set, a configuration was chosen which had good generic performance, while also being close to the minimum run time. The resulting compile configuration was:

```
-DHAVE_PTHREAD -DNO_ASSERT_CHECKS -DUSE_TLS -DBLOCKS_FORMULA=1024
-DCHUNK_POWER=2 -DALLOCATION_SIZE_RATIO=1080 -DCB_SIZE_BITS=3
-DNO_BLOCK_ORDERING -DMIN_ALIGN=4 -DESTIMATED_MIN_BLOCKS=3
```

## Environment Variables

DFWMALLOC_STARTUP_MESSAGE

Unset for disabled. Print a startup message with the process ID. This feature is enabled with -DWITH_BTANDCHECK.

DFWMALLOC_BACKTRACE

Unset for disabled. Store a call-trace against each malloc with verification support. This feature is enabled with -DWITH_BTANDCHECK. See **Back-tracing and Verification Support**.

DFWMALLOC_FRAGTOOL

Unset for disabled. Allows dumping of the current allocation structure for inspection. See **Inspecting Fragmentation**.

## Support for Posix Threads

A new arena is created when a thread calls malloc for the first time. If a thread exits, the arena is flagged as out-of-use and destroyed, but only if there are zero unfreed pointers. When the last unfreed pointer is free'd then DFWMalloc destroys the arena. Arenas that are stale (due to a leak) are permitted to be safely reused by the priority of the oldest out-of-use arena first. This strategy minimizes the impact of threads terminating with leaks.

In order to handle the creation and destruction of arenas, DFWMalloc intercepts certain C Library functions like pthread_create(). This interception is done in lieu of building DFWMalloc into the C Library and having those functions directly implement creation and destruction of arenas. Interception does not appear to cause any problems. It is hoped that the GNU C Library can implement hooks specific to the requirements of a pluggable allocation library.

If more threads are created than available arenas, then DFWMalloc writes to stderr and returns NULL for the respective operation, setting errno to ENOMEM.

## C++ Support

The following C++ operators are supported. This includes C++17 operators:

```
void *operator new (size_t size);
void *operator new[] (size_t size);
void *operator new (size_t size, std::align_val_t align);
void *operator new[] (size_t size, std::align_val_t align);
void *operator new (size_t size, std::nothrow_t const &);
void *operator new[] (size_t size, std::nothrow_t const &);
void *operator new (size_t size, std::align_val_t align, std::nothrow_t const &);
void *operator new[] (size_t size, std::align_val_t align, std::nothrow_t const &);
void operator delete(void*);
void operator delete[](void*);
void operator delete(void*, std::size_t);
void operator delete[](void*, std::size_t);
void operator delete(void*, const std::nothrow_t&);
void operator delete[](void*, const std::nothrow_t&);
void operator delete(void*, std::align_val_t);
void operator delete(void*, std::align_val_t, const std::nothrow_t&);
void operator delete[](void*, std::align_val_t);
void operator delete[](void*, std::align_val_t, const std::nothrow_t&);
void operator delete(void*, std::size_t, std::align_val_t);
void operator delete[](void*, std::size_t, std::align_val_t);
```

## Back-tracing and Verification Support

In order to serve as both a debugging tool as well as a replacement high-performance malloc library, DFWMalloc can be built in multiple ways.

*A. Pure debugging tool*

Build options:

-DUSE_TLS -DWITH_BTANDCHECK -DWITH_ACCOUNTS -DNO_CB_HEAPS

In this mode, DFWMalloc verifies every allocation of memory overwrite, as well as performing double-free detection. DFWMalloc also accounts for every byte allocated. This accounting data is available via the DFWMalloc API. Verification steps are done on free(), realloc(), delete, and dfwmalloc_check().

If the environment variable DFWMALLOC_BACKTRACE backtrace is set, then all allocations and free's are traced and a duplicate free will dump a hex pointer call-trace log detailing both the allocate, the first free, and the second free. This has an order-of-magnitude performance penalty. The call-trace can be decoded with the **addr2line** utility. In order to print meaningful hex pointers the following steps are necessary:

1. Your program must be compiled and linked with the compiler options: -no-pie -g -Og (-Og is optional but preferred).
2. Run the command: sysctl -w kernel.randomize_va_space=0

Without these steps the addr2line command may print ??:0 for all addresses.

Example addr2line usage is as follows:

```
addr2line -e my_exe   0x419b54 0x419bb3 0x419bda 0x419c1a 0x7ffff75e2b97 0x40101a
```

*B. Dual-use debugging tool and allocator*

Build options:

-DUSE_TLS -DWITH_BTANDCHECK -DWITH_ACCOUNTS

In this mode, DFWMalloc performs high-speed CB allocations for small blocks and verified allocations for large blocks. If the environment variable DFWMALLOC_BACKTRACE is set then all allocations are regular allocations with double-free and overwrite detection and call-trace logging on errors.

Here is an example under the bourne shell:

```
DFWMALLOC_STARTUP_MESSAGE="gzip running under DFWMalloc" \
  DFWMALLOC_BACKTRACE=1 \
    LD_PRELOAD=/usr/local/lib/libdfwmalloc-debug.so \
      /bin/gzip -h
```

Even without DFWMALLOC_BACKTRACE set, this mode suffers a moderate performance penalty due to the many *assert* checks at all steps of allocation management.

*C. High performance allocator*

Build options:

-DUSE_TLS -DHAVE_PTHREAD

In this mode, verification steps are performed to check for corruption of internal data structures. This has a small performance penalty.

*D. Maximum performance allocator*

Build options:

-DUSE_TLS -DHAVE_PTHREAD -DNO_ASSERT_CHECKS

In this mode, no verification steps are performed. This mode should be used for stable production software desiring the highest performance.

## List of Verification Errors

Following is a list of errors that DFWMalloc will write to your configured log callback function, or, if unset, to stderr. DFWMalloc will log the error and then call abort().

*corrupted block on free list (truncate heap)*

When the mmap'd region was released back to the operating system, a block was found that had its head or tail watermarks overwritten.

*user wrote to freed block*

During a reuse of a previously free'd block, it was detected that the calling program wrote to the block after having free'd it.

*corrupted block on free list*

During a reuse of a previously free'd block, it was detected that the calling program overwrote past the head or tail of the block.

*caller trying to realloc already-freed block*

The caller performed a free then a realloc on a pointer.

*corrupted block on realloc*

The caller performed a realloc on a block after writing past the head or tail of the block.

*corrupted block found on free (shared)*

The caller free'd a block in another thread after writing past the head or tail of the block.

*caller trying to free already-freed block*
*caller trying to free already-freed block (delayed)*

The caller called free twice on a block (so called "double-free" programming error). A delay means this was only discovered when the original thread came around to this block.

*corrupted block found on free*

The caller free'd a block after writing past the head or tail of the block.

## DFWMalloc API Usage

The DFWMalloc API provides extended functionality beyond the standard C Library malloc. This section gives some examples. See the respective header files for function prototypes:

Check that a pointer is valid, returns the allocated size, checks the bounds for corruption:

```
dfwmalloc_check(...)
```

To dump leaks use:

```
dfwmalloc_dump_leaks(...)  or
dfwmalloc_log_leaks()
```

If a foreign thread free'd a lot of stuff on your behalf, then catch up with:

```
dfwmalloc_free_foreign()
```

To free mapped pages do this call once every second:

```
dfwmalloc_truncate_heap()
dfwmalloc_dfwtruncate_heap(...)
```

To create your own arena in which to malloc and free, do:

```
f = dfwmalloc_new ();
p = dfwmalloc_malloc (f, c);
dfwmalloc_free (f, p);
if (0) free (p); // <== you can also free from a different arena but you
                 // incur a performance penalty.
leaks = dfwmalloc_destroy (f);
assert (leaks == 0);
f = NULL;
```

These functions are for getting usage stats:

```
dfwmalloc_stats(...)
dfwmalloc_heap_size()       <-- simpler version of dfwmalloc_stats
dfwmalloc_alloced_bytes()   <-- simpler version of dfwmalloc_stats
```

To dump all control blocks to the log for fragmentation assessment. See **Inspecting Fragmentation** for more info:

dfwmalloc_efficiency()
dfwmalloc_dfwefficiency(...)

Get the version as (major << 20) | (minor << 10) | patch

dfwmalloc_version()
DFWMALLOC_VERSION

Also note:

strings lib*.so | grep VERSION
DFWMALLOC_VERSION_MAJOR
DFWMALLOC_VERSION_MINOR
DFWMALLOC_VERSION_PATCH

See also as follows for disabling built-in malloc libraries,

https://fuzzing-project.org/tutorial-malloc.html
https://web.archive.org/web/20191008001924/https://fuzzing-project.org/tutorial-malloc.html

## Inspecting Fragmentation

This feature is not compatible with the compile option -DNO_CB_HEAPS since only control blocks' fragmentation are logged.

The API functions dfwmalloc_efficiency() and dfwmalloc_dfwefficiency() will dump the fragmentation structure to the log (or stderr if no log callback function is set). Alternatively, the program dfwmalloc-fragtool exists for dumping the fragmentation structure of an arbitrary process on the system. Running dfwmalloc-fragtool is equivalent to calling dfwmalloc_efficiency() within your program. (dfwmalloc-fragtool uses POSIX real-time signal SIGRTMIN + 5. If another program or library simultaneously uses this signal, this will result in undefined behavior.)

To use dfwmalloc-fragtool, the procedure is as follows:

1. Compile the dfwmalloc shared library with the -DWITH_FRAGTOOL option or use the by-default built library libdfwmalloc-fragtool.so.
2. Set the environment variable DFWMALLOC_FRAGTOOL=1
3. Run your program linked with libdfwmalloc, and find out its Process-ID *pid*.
4. Run dfwmalloc-fragtool *pid*
5. Run dfwmalloc-svg to create a graphical tiled representation of the results.

Example usage:

```
#> DFWMALLOC_FRAGTOOL=1 LD_PRELOAD=./libdfwmalloc-fragtool.so sleep 100 &
#> echo $!
54321
#> ./dfwmalloc-fragtool 54321 > example.txt
#> ./dfwmalloc-svg example.txt > example.svg
#> geeqie example.svg
```

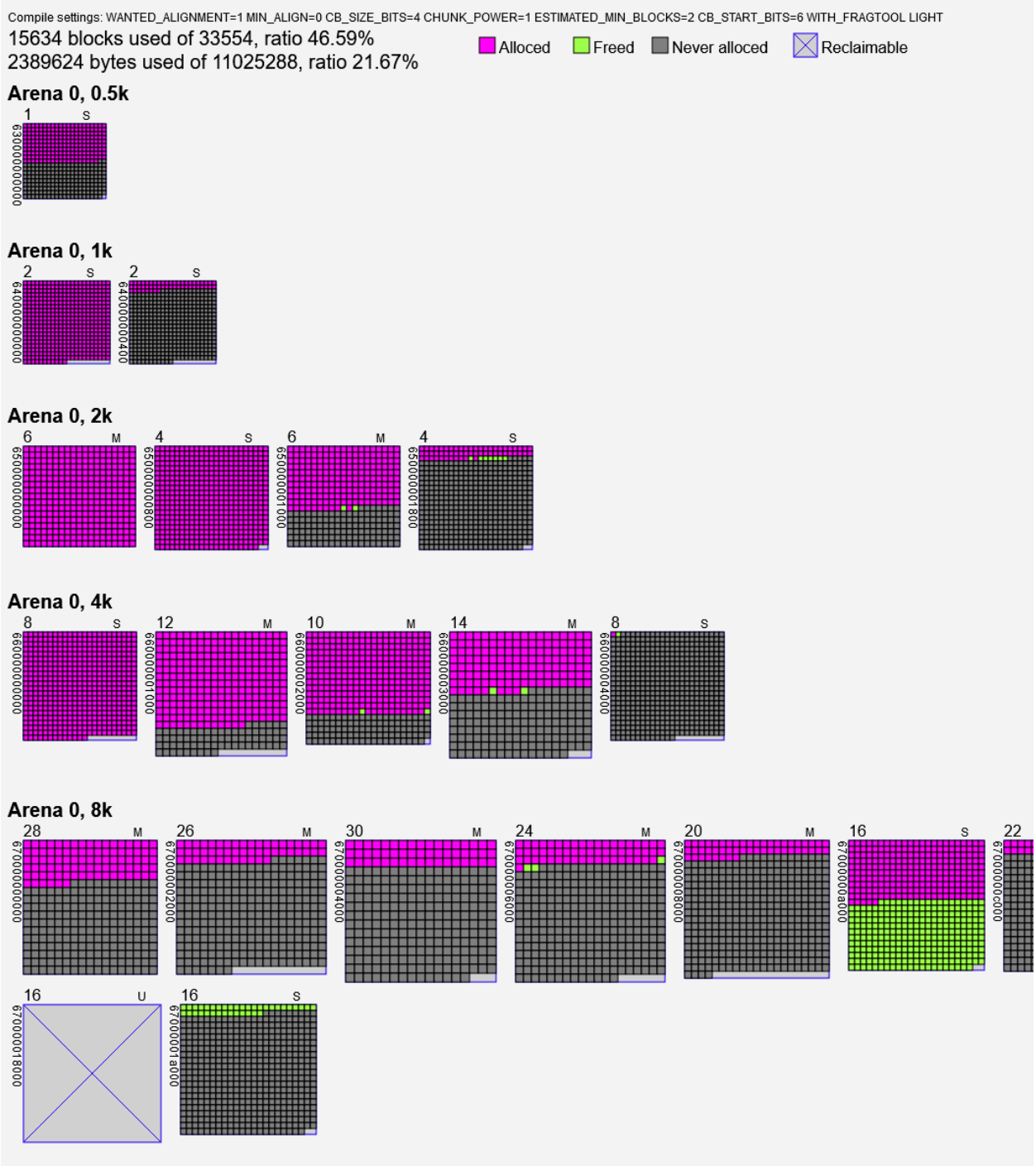
An example snippet of output is shown here:

```
Starting dump of fragmentation data
Compile settings: WANTED_ALIGNMENT=1 MIN_ALIGN=0 CB_SIZE_BITS=4
  CHUNK_POWER=1 ESTIMATED_MIN_BLOCKS=2 CB_START_BITS=6 WITH_FRAGTOOL LIGHT
output version 2
arena heap cb size n used freed total mgmnt packing usedmask
0 3 630000000000 512 1 208 15 398 112 S ffffffffffffffffffffffffffffffff
0 4 640000000000 1024 2 451 0 451 120 S ffffffffffffffffffffffffffffffff
0 4 640000000400 1024 2 52 3 451 120 S fffffffffffff000
0 5 650000000000 2048 6 323 0 323 104 M ffffffffffffffffffffffffffffffff
0 5 650000000800 2048 4 481 0 481 120 S ffffffffffffffffffffffffffffffff
0 5 650000001000 2048 6 200 2 323 104 M ffffffffffffffffffffffffffffffff
0 5 650000001800 2048 4 57 7 481 120 S ffffffffffffff40
0 6 660000000000 4096 8 495 1 496 120 S fffffffffffffffffffffffffffffff7
```

The fields have the following meaning:

1, arena: The arena number in the range of 0 to ≈1<<N_ARENAS_BITS. This can be interpreted as distinguishing a separate thread.
2, heap: The heap index in the range of 0 through 5 (depending on compile options).
3, cb: A pointer to the start of the control block.
4, size: The size in bytes of the control block.
5, n: The size of a single allocation.
6, used: The number of used blocks.
7, freed: The number of previously used blocks that were freed.
8, total: The sum of used and available blocks. (n x total + mgmnt = size)
9, mgmnt: Bytes of management data at start of control block.
10, packing: See **Packing Styles**. U means an unused Control Block.
11, usedmask: Bitmask of used blocks in hexadecimal format.

Example SVG output for a single-threaded GUI application:



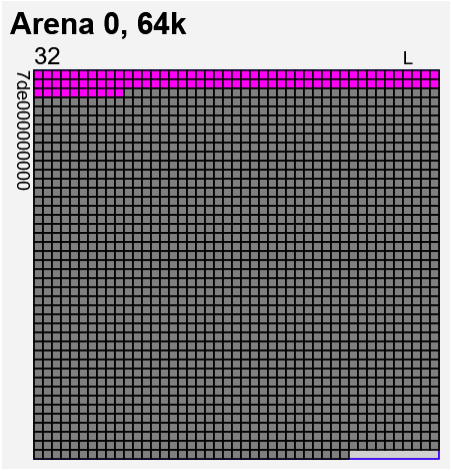
## Crypto Mode

The compile options -DWITH_CRYPTO -DNO_BLOCK_ORDERING enable cryptographic randomization of allocated blocks within a control block. (At the time of writing, the randomization algorithm is designed for performance and is not *cryptographically strong*. The user is invited to implement their own dfwmalloc_get_random_cb_int() function if a strong algorithm is required.) Randomizing blocks in this way will make certain types of hacking attacks on programs more difficult.

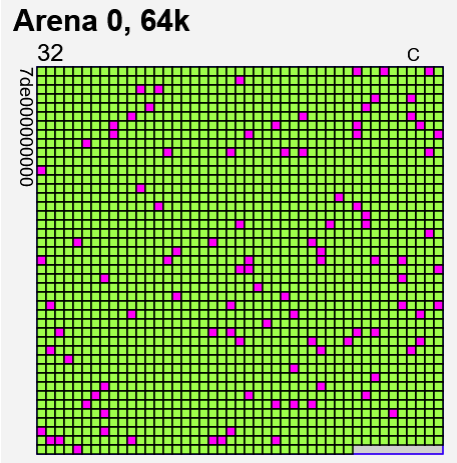
To illustrate the effect of this, consider the following program:

```
for (j = 0; j < 100; j++)
   p[j] = malloc (32);
```

The fragool output without the crypto feature is as follows:




The fragool output with the crypto feature enabled is as follows (lime tiles show “freed” blocks):




## Inline Malloc for Extreme Performance

For improved speed over the Standard C Library API, DFWMalloc has an inlining option to further reduce clock cycles.

Consider a program that has a pattern of usage that allocates lots of blocks of a *known* size. This pattern allows inlining of malloc calls, provided the program has made preparations at startup time and include the header *dfwmalloc_inline.h*. The API call to prepare to malloc a particular size is,

```
int dfwmalloc_prepare (struct malloc_calc *precalc, size_t n);
```

Calls to malloc() and free() can be substituted with these two inline functions,

```
static inline void *malloc_precalc (struct malloc_calc *precalc);
static inline void free_precalc (struct malloc_calc *precalc, void *v);
```

Here is an example,

```
#include <dfwmalloc_inline.h>

struct malloc_calc precalc;

void run_program() {
        ...
        struct my_widget *p;
        p = malloc_precalc (&precalc);
```

```
        …
        free_precalc (&precalc, p);
}

int main() {
        int r = dfwmalloc_prepare (&precalc, sizeof(struct my_widget));
        assert (!r);
        run_program();
}
```

## C++ Inline *new/delete* for Extreme Performance

C++ allows inlining analogous to C using templates. Here is an example:

```
// my_class.h

#include "dfwmalloc_cxx.h"

extern dfwmalloc_class my_object;

class my_class : public malloc_object<dfwmalloc_class, my_object>
{
  public:
    my_class();
    ~my_class();
    void my_method1();
    void my_method2();
    void my_method3();

  private:
    int my_var1;
    int my_var2;
    int my_var3;
};
```

Using the above declarations allows the programmer to perform *new* and *delete* operations on the class *my_class* and expect that memory allocation and deallocation is done as *inline operations*. For example,

```
// my_class.cc

# include “my_class.h”

dfwmalloc_class my_object(sizeof (my_class));
```

```
run_program () {
        my_class *o = new my_class;
        …
        delete o;
}
```

## Compiled-in vs generated static data

DFWMalloc tries to retain the one-source-file philosophy to make for trivial insertion into any existing code base. To facilitate this functionality, DFWMalloc must generate certain tables of static data on startup. On the other hand, a system-wide shared library should not do this, but instead generate such data at compile time. This alternative is also supported. The macros USE_PRECOMPILED_STATIC_DATA and PRECOMPILE_STATIC_DATA are applicable to this case. See the Makefile for an example of how to use these macros to make two compile steps.

## Bugs

1. No testing has been done for 32-bit systems. This is currently 64-bit only, although much effort has been made to ensure portability. A 32-bit port should be low-effort, but functionality will be reduced. Not all features have seen production testing.

2. Repeated calls to malloc(0) return the identical pointer. Duplicate free's of this pointer are allowed at all times.

3. No mallopt() support. Minimal malloc_info() support -- total consumed bytes only.

4. malloc_trim() ignores its argument and does a full trim, and always returns 1.

5. If N is 1<<WANTED_ALIGNMENT (the alignment) then malloc(n<N) returns a pointer with alignment of the highest power of 2 less than n. This does not seem to create any problems.

6. Not a bug, but one should note the discrepancy between POSIX and C++ specifications. POSIX does not require 16-byte alignment, but C++ does. 16-byte alignment is the default.

7. With the default compile options, a single process can create no more than ≈1000 concurrent threads.

8. The inline new/delete mechanism has not been tested within a real application.

9. The crypto randomization algorithm is not cryptographically strong.

## Compile Options

```
/* Configuration: */

/* Basic configuration: */

/* Guaranteed alignment of all malloc() return values (power of 2).
 * Example alignment requirements for 64-bit architectures:
 *   Intel CPU: 1<<0 = 1
 *   Most RISC: 1<<3 = 8
 *   C: 1<<3 = 8
 *   C++: 1<<4 = 16
 *   Chrome: 1<<5 = 32 */
/* #undef WANTED_ALIGNMENT */
/* #define WANTED_ALIGNMENT 4 */

/* Add backtrace support and bounds checking. Call dfwmalloc_enable_backtrace() to enable: */
/* #undef WITH_BTANDCHECK */
/* #define WITH_BTANDCHECK */

/* Add accounting: */
/* #undef WITH_ACCOUNTS */
/* #define WITH_ACCOUNTS */

/* Add support for fragmentation logging tool dfwmalloc-fragtool: */
/* #undef WITH_FRAGTOOL */
/* #define WITH_FRAGTOOL */

/* Use crypto-randomized blocks: */
/* #undef WITH_CRYPTO */
/* #define WITH_CRYPTO */

/* For multithreading support: */
/* #undef HAVE_PTHREAD */
/* #define HAVE_PTHREAD */

/* For thread-local-storage support: */
/* #undef USE_TLS */
/* #define USE_TLS */

/* Do not even do basic sanity checks. Applicable only when WITH_BTANDCHECK is undefined: */
/* #undef NO_ASSERT_CHECKS */
/* #define NO_ASSERT_CHECKS */

/* Never truncate the heap: */
/* #undef NO_HEAP_TRUNCATION */
/* #define NO_HEAP_TRUNCATION */
```

```
/* malloc size is 32 bits: */
/* #undef LIGHT */
/* #define LIGHT */

/* User can define the maximum expected allocation: */
/* #undef USER_MAX_MALLOC */
/* #define USER_MAX_MALLOC (? * 1024 * 1024) */

/* User can define the granularity of increasing bucket sizes for regular allocations: */
/* #undef BUCKET_SMOOTHNESS */
/* #define BUCKET_SMOOTHNESS 6 */

/* Don't use control blocks: all allocations are Regular allocations: */
/* #undef NO_CB_HEAPS */
/* #define NO_CB_HEAPS */

/* Do not use the lock-free queue mechanism for allocating in one thread and free'ing in another: */
/* #undef NO_LOCKFREEVISITOR */
/* #define NO_LOCKFREEVISITOR */

/* Do not trap the pthread_exit() function. This means you have to call
 * dfwmalloc_pthread_exit() on thread exit. Saves you have to link with -ldl: */
/* #undef NO_PTHREAD_EXIT_OVERRIDE */
/* #define NO_PTHREAD_EXIT_OVERRIDE */



/* Less common configuration: */

/* This is used to not do certain operations too often. Interpreted as 1<<N MHz (2048 for N=11). It doesn't matter if it
off by a small factor: */
/* #undef ROUGH_CPU_SPEED_MHZ */
/* #define ROUGH_CPU_SPEED_MHZ 11 */

/* Roughly the number of threads you can create, 1024 = 1<<10: */
/* #undef N_ARENAS_BITS */
/* #define N_ARENAS_BITS 10 */

/* Do not provide the exported interface functions like dfwmalloc_new()/dfwmalloc_destroy() for creating user
dfw-arenas: */
/* #undef NO_INTERNAL_EXPOSE */
/* #define NO_INTERNAL_EXPOSE */

/* Utility function: */
/* #undef WANT_PRINTFSIMPLE */
/* #define WANT_PRINTFSIMPLE */

/* Fidelity checks: */
/* #undef WITH_UNITTESTS */
/* #define WITH_UNITTESTS */
```

```
/* Disable inlining optimizations: */
/* #undef DISABLE_INLINE */
/* #define DISABLE_INLINE */

/* Minimum alignment for small areas. This allows DFWMalloc to use a shift to increase the potential size of a chunk:
*/
/* #undef MIN_ALIGN */
/* #define MIN_ALIGN 3 */

/* Number of bits in pointers reserved for the heap identifier: */
/* #undef CB_SIZE_BITS */
/* #define CB_SIZE_BITS 3 */

/* Steps increase of chunk sizes. 1 means 1024,2048,4096,... and 2 means 1024,4096,16384,...  */
/* #undef CHUNK_POWER */
/* #define CHUNK_POWER 2 */

/* Number of blocks in the largest control block. Used as 1<<n: */
/* #undef ESTIMATED_MIN_BLOCKS */
/* #define ESTIMATED_MIN_BLOCKS 4 */

/* How to estimate the number of blocks per control block for a particular size of malloc: */
/* #undef BLOCKS_FORMULA */
/* #define BLOCKS_FORMULA        (500000 / (n + 1000) + 32) */

/* Do not order small blocks within their control chunk. This has a slight performance advantage,
 * but hurts cache usage. It will also limit the usable size of control block for packing reasons: */
/* #undef NO_BLOCK_ORDERING */
/* #define NO_BLOCK_ORDERING */

/* Use bitwise packing when possible: */
/* #undef COMPACT_MODE */
/* #define COMPACT_MODE */

/* Ratio of successive allocation sizes for Control Block allocations: */
/* #undef ALLOCATION_SIZE_RATIO */
/* #define ALLOCATION_SIZE_RATIO 1100 */

/* Size of control blocks of the heap with index 0. Used as 1<<n: */
/* #undef CB_START_BITS */
/* #define CB_START_BITS 10 */

/* Needed for Chrome or any application that doesn't like its exit/_exit/_Exit functions
 * being overridden. This does not affect atexit() which is still used: */
/* #undef NO_OVERRIDE_EXIT_FUNCTIONS */
/* #define NO_OVERRIDE_EXIT_FUNCTIONS */

/*
 *
```

```
 * Extra address ranges to avoid. The "END" parameter must be defined is the last byte plus 1,
 * for example define the range as 0x38000000000-0x40000000000 and not 0x38000000000-0x3FFFFFFFFFF.
 *
 * #define AVOID_ADDRESS_RANGE_1_START    0x000000000000UL
 * #define AVOID_ADDRESS_RANGE_1_END      0x000000000000UL
 * #define AVOID_ADDRESS_RANGE_2_START    0x000000000000UL
 * #define AVOID_ADDRESS_RANGE_2_END      0x000000000000UL
 * #define AVOID_ADDRESS_RANGE_3_START    0x000000000000UL
 * #define AVOID_ADDRESS_RANGE_3_END      0x000000000000UL
 * #define AVOID_ADDRESS_RANGE_4_START    0x000000000000UL
 * #define AVOID_ADDRESS_RANGE_4_END      0x000000000000UL
 * #define AVOID_ADDRESS_RANGE_5_START    0x000000000000UL
 * #define AVOID_ADDRESS_RANGE_5_END      0x000000000000UL
 *
 */


/* End configuration. */
```

## Source Code Metrics

### *System calls*

DFWMalloc uses the following C Library calls and Unix system calls:

| | | |
|---|---|---|
| abort | memcpy | pthread_mutex_lock |
| backtrace | memset | pthread_mutex_unlock |
| dlsym | mmap | strcat |
| exit | munmap | strcpy |
| _exit | pthread_attr_getdetachstate | strlen |
| _Exit | pthread_create | write |
| getenv | pthread_exit | writev |
| getpid | pthread_join | |
| memcmp | pthread_mutex_init | |

### *Non-portable clauses*

__thread
__attribute__(...)
__builtin_clzl()
asm volatile ("rdtsc":"=a","=d");
__sync_bool_compare_and_swap()
__sync_synchronize()
__sync_fetch_and_add()

### *Lines of code*

| | |
|---|---|
| Minimal source without call-trace features: | 2.7k |
| Full source: | 7.3k |

*Compliance*

C90 through C99
C++98 through C++-17
POSIX-1997/SUSEv2 and later

*Default excluded address space*

0000 0000 0000 - 0080 0000 0000
5500 0000 0000 - 5680 0000 0000
7E00 0000 0000 - 8000 0000 0000

*Ports*

Linux x86_64
FreeBSD amd64

*Coding style*

K&R

## Building for the Chrome Web Browser

A patch to allow Chrome to use an external memory allocator is included in the DFWMalloc source (chrome-patch.diff). You will need to build Chrome using the default Google directions: https://www.chromium.org/developers/how-tos/get-the-code

Note that Chrome is purported to have additional internal memory allocators that will not be overridden. Thus this procedure will only implement part of Chrome's memory allocation requirements.

The following DFWMalloc compile options are mandatory for Chrome:

```
-DWANTED_ALIGNMENT=5
-DHAVE_PTHREAD
-DUSE_TLS
-DNO_OVERRIDE_EXIT_FUNCTIONS
```

The following compile options are optional. These options will allow you to see the DFWMalloc arenas being created and reused:

```
-DN_ARENAS_BITS=7
-DLOG_ARENAS
```

The makefile Makefile.chrome is provided.

After compilation is complete, use the following shell command to run Chrome,

LD_PRELOAD=/*path*/libdfwmalloc-debug.so ./out/Default/chrome

To see all leaks on exit run,

DFWMALLOC_BACKTRACE=1 LD_PRELOAD=/*path*/libdfwmalloc-debug.so ./out/Default/chrome

## Appendix A

### Discussion of Performance Benchmarking

Benchmark results of all malloc libraries appeared to vary widely with: range of block sizes, total memory consumed, and size of blocks. To understand these differences, consider how a benchmark might distribute a range of block sizes. A distribution could vary constantly or have a fixed set of sizes; it could perform more allocations for large blocks or smaller blocks; and it could use completely arbitrary allocation sizes or restricted allocation sizes to a small number of predefined sizes in the manner of a known application. Yet further options are linear versus logarithmic distribution of block sizes and number of blocks. Other considerations are read and write access of blocks in a similar pattern to a real application.

All these considerations have large impacts on storage efficiency and speed. Varying the pattern of the test is instructive for comparing DFWMalloc to other libraries. In all contrived scenarios DFWMalloc had either better speed or better memory consumption. In no scenario did DFWMalloc show runaway behavior.

Our benchmarking reveals that JEMalloc is the closest competitor to DFWMalloc, whereas all other allocators had poor performance for at least one contrived load scenario. Legacy allocators were found to have order-of-magnitude worse performance and consequently were omitted from the results.

**Benchmark t-test1.c source code changes**

The *t-test1.c* test program was modified as follows for the test under **Performance** above. These changes were chosen to be minimal, essentially the only performance-impacting change was to force mapping of the block into memory by executing a single access to the block:

```
--- t-test1.c
+++ t-test1.c
@@ -162,7 +162,12 @@
 #include <sys/wait.h>
 #endif

+#include <unistd.h>
 #include <sys/types.h>
+#include <sys/stat.h>
+#include <string.h>
+#include <fcntl.h>
+#include <errno.h>
 #include <malloc.h>


@@ -208,6 +213,18 @@

 #endif

+/* force the page to be mapped */
+static void mem_page(unsigned char *ptr, size_t size)
+{
+         size_t i;
+
+         if (!size) return;
+         for (i = 0; i < size; i += 4096)
+         {
+                  ptr[i] = (unsigned char) 0xAA;
+         }
+}
+
 #if TEST > 0

 static void mem_init(unsigned char *ptr, size_t size)
@@ -241,7 +258,7 @@

 static int zero_check(void *p, size_t size)
 {
-         unsigned *ptr = p;
+         unsigned *ptr = (unsigned *) p;
          unsigned char *ptr2;

          while (size >= sizeof(*ptr))
@@ -280,13 +297,13 @@
          {
                   /* memalign */
                   if (m->size > 0) free(m->ptr);
-                  m->ptr = memalign(sizeof(int) << r, size);
+                  m->ptr = (unsigned char *) memalign(sizeof(int) << r, size);
          }
          else if (r < 20)
          {
                   /* calloc */
                   if (m->size > 0) free(m->ptr);
```

```
-                   m->ptr = calloc(size, 1);
+                   m->ptr = (unsigned char *) calloc(1, size);
 #if TEST > 0
                    if (zero_check(m->ptr, size))
                    {
@@ -304,13 +321,13 @@
          {
                    /* realloc */
                    if (!m->size) m->ptr = NULL;
-                   m->ptr = realloc(m->ptr, size);
+                   m->ptr = (unsigned char *) realloc(m->ptr, size);
          }
          else
          {
                    /* malloc */
                    if (m->size > 0) free(m->ptr);
-                   m->ptr = malloc(size);
+                   m->ptr = (unsigned char *) malloc(size);
          }
          if (!m->ptr)
          {
@@ -319,6 +336,7 @@
          }

          m->size = size;
+         mem_page(m->ptr, m->size);
 #if TEST > 0
          mem_init(m->ptr, m->size);
 #endif
@@ -348,12 +366,12 @@
          size_t size, bins;
 };

-#if TEST > 0
+#if TEST > 1
 static void bin_test(struct bin_info *p)
 {
          int b;

-         for (b = 0; b < p->bins; b++)
+         for (b = 0; b < (int) p->bins; b++)
          {
                    if (mem_check(p->m[b].ptr, p->m[b].size))
                    {
@@ -379,7 +397,7 @@
 static void malloc_test(void *ptr, size_t stack_len)
 #endif
 {
-         struct thread_st *st = ptr;
+         struct thread_st *st = (struct thread_st *) ptr;
          int i, pid = 1;
          unsigned b, j, actions;
          struct bin_info p;
@@ -406,7 +424,7 @@
          }
 #endif

-         p.m = malloc(st->bins * sizeof(*p.m));
+         p.m = (struct bin *) malloc(st->bins * sizeof(*p.m));
          p.bins = st->bins;
          p.size = st->size;
          for (b = 0; b < p.bins; b++)
@@ -510,6 +528,69 @@
          return 0;
 }

+
+#ifdef GOOGPERF
+struct ProfilerOptions;
+
+int ProfilerStart(const char* fname);
+int ProfilerStartWithOptions(const char *fname, const struct ProfilerOptions *options);
+void ProfilerStop(void);
+#endif
+
+
+static long find_field (const char *field, const char *buf)
+{
```

```
+    const char *p;
+    p = strstr (buf, field);
+    if (!p)
+        return -1;
+    p += strlen (field);
+    while (*p && *p <= ' ')
+        p++;
+    if (!*p)
+        return -1;
+    return atol (p);
+}
+
+static long get_os_heap_size (void)
+{
+    char f[256];
+    char buf[65536];
+    int buflen = 0;
+    int fd;
+    snprintf (f, sizeof (f), "/proc/%ld/status", (long) getpid ());
+    fd = open (f, O_RDONLY);
+    if (fd < 0) {
+        perror (f);
+        exit (1);
+    }
+    for (;;) {
+        int r;
+        r = read (fd, buf + buflen, sizeof (buf) - buflen);
+        if (r <= 0)
+            break;
+        buflen += r;
+    }
+    close (fd);
+    buf[sizeof (buf) - 1] = '\0';
+    return find_field ("VmRSS:", buf);
+}
+
+static long max_heap = 0L;
+static int heap_thread_running = 1;
+static void *heap_thread (void *dummy)
+{
+    (void) dummy;
+    while (heap_thread_running) {
+        long heap_size;
+        usleep (100000);
+        heap_size = get_os_heap_size ();
+        if (max_heap < heap_size)
+            max_heap = heap_size;
+    }
+    return NULL;
+}
+
 int main(int argc, char *argv[])
 {
          int i, bins;
@@ -517,6 +598,14 @@
          int i_max = I_MAX;
          size_t size = MSIZE;
          struct thread_st *st;
+         pthread_t heap_thread_id;
+
+#ifdef GOOGPERF
+         ProfilerStart("t-test1");
+#endif
+
+         pthread_create(&heap_thread_id, NULL, heap_thread, NULL);
+

          if (argc > 1) n_total_max = atoi(argv[1]);
          if (n_total_max < 1) n_thr = 1;
@@ -547,7 +636,7 @@
          printf("total=%d threads=%d i_max=%d size=%ld bins=%d\n",
                      n_total_max, n_thr, i_max, size, bins);

-         st = malloc(n_thr * sizeof(*st));
+         st = (struct thread_st *)  malloc(n_thr * sizeof(*st));
          if (!st) exit(-1);

 #if !defined NO_THREADS && defined __sun__
```

```
@@ -573,7 +662,7 @@
                                n_thr = i;
                                break;
                        }
-                       printf("Created thread %lx.\n", (long)st[i].id);
+//                     printf("Created thread %lx.\n", (long)st[i].id);
                }

                for (n_running = n_total = n_thr; n_running > 0;)
@@ -645,9 +734,19 @@
                        if (st[i].sp) free(st[i].sp);
                }
                free(st);
+
+#ifdef GOOGPERF
+               ProfilerStop();
+#endif
+
 #if USE_MALLOC
                malloc_stats();
 #endif
+
+               heap_thread_running = 0;
+               pthread_join (heap_thread_id, NULL);
+
+               printf("max memory: %ld\n", max_heap);
                printf("Done.\n");
-               return 0;
+               exit (0);
 }
```