Optimizing the Compiler's Memory Usage? Let Us Implement a Basic Profiler First!

Gunnar Kudrjavets
University of Groningen
g.kudrjavets@rug.nl
@gunnarku

Aditya Kumar
ACM Distinguished Speaker
hiraditya@msn.com
@_hiraditya_

Jeff Thomas
Cyclist, mountaineer, skier
jeffdthomas@gmail.com
@ jeffdthomas

TL;DR

- Implementing a custom profiling solution is a complex problem for niche products.
- While solving individual subtasks can be an exciting intellectual pursuit, implementing a working solutions comes down to pragmatic trade-offs.
- Systems programmer's reality is unpleasant.

Speaker bio aka I’m not a compiler engineer

- 2000 —2015, Microsoft Corporation
  - Redmond, WA, USA. Shifting bits.
- 2015 —2020, Facebook, Inc. (aka Meta Platforms, Inc.)
  - Seattle, WA, USA. Shifting more bits.
- 2020 —2021, The “Eat, Pray, Love” phase of life
  - Colorado, Nepal, Oregon, getting into trouble.
- 2021 —2023, Writing random research papers and finishing the dissertation.
  - Hapless PhD candidate.

Why should compiler engineers care?

- A philosophical meta question: is optimizing compiler resource usage a niche problem?
- Organizations and projects with large code bases (e.g., Linux, Microsoft Windows, Mozilla) deeply care.
- Build is never sufficiently fast.

Typical solutions

- Delete dead code - who does that?
  - "Pruning and Polishing: Keeping OpenBSD Modern"
- Hardware (e.g., buying better SDDs) can help only so much.
- Parallelization reaches its limits (OS limits, overall resource usage per whatever container the build runs in, build rules, etc.).
- The inevitable conclusion - the "lighter" your tools, the better.

Problem statement we extrapolate from

- **Context**
  - A user mode process with a relatively short lifetime compared to for example a daemon/service.
- **Goals**
  - Reduce the clock time (wall time) of specific tasks (e.g., a function).
  - Understand where the amount of "work" has increased in production.
- **Problem**
  - Reliable measuring and profiling of short periods (e.g., why a scenario that took 53 ms now takes 81 ms?) is susceptible to Heisenberg effect ("the very act of measurement or observation directly alters the phenomenon under investigation").
- Method
  - Use the memory allocator churn as a proxy metrics.
  - Yes, there's also CPU usage, I/O (disk, network, IPC/RPC/XPC).

Core principles: ability to profile in production

- Map is not a territory.
- Internal test environments, build labs, dogfooding, etc. can only provide you with an approximation of reality.
- Dogfooding data is often biased. For example, how many FAANG employees still use an iPhone 6 or MacBook from 2015 or are on 3G networks?
- In general, "you have no clue what people are running and how."
  - Prime example: backwards compatibility in OS.
- Design decision: whatever we do—it must be seamless.

Core principles: no new build type required

- The build matrix for each nontrivial project is significant.
- Getting all builds passing is like a game of Whac-a-Mole.
- **Platform** (x64, ARM32, ARM64, ...) x **build flavor** (debug, release, ship, ...) x **sanitizers** (ASan, TSan, UBSan, ...) x **compiler** (Clang, GCC, MSVC, ...) x **build environment** (Bazel, Buck, Ninja, Makefile, ...) x **OS** (Linux, macOS, Windows, ...) x . . .
- Design decision: we will not add to this madness.

Core principles: profiling must be on-demand

- Cool KidZ practice Continuous Profiling.
- We are not interested in "profiling everything all the time."
- Clearly, we cannot deploy special tools like (e.g., BPF, kerntrace, perf) with our product.
- Must have an ability to control profiling functionality using feature flags depending on certain criteria.
- Cannot ask users "Hey, would you run Clang in the profile mode for us?"

Epistemological humility in engineering

- Most problems have already been solved in some shape or form in the past.
- Lots of lessons from other engineers floating around.
- Need to be aware of our own biases as engineers.
  - For an operating systems engineer, every problem can be solved by developing a new memory manager, optimizing a locking scheme, or tweaking a spin-lock.
- There will be a lots of “temptations” on the way.

Sample guidance #1

"Memory management is a solved problem. Why don't you just rewrite everything in OCaml and be done with the problem forever?"

Sample guidance #2

"Just throw some ML model on it.
Talk to data science people."

Sample guidance #3

"This is a hard problem.
You are doomed."

(Polite version of what was said.)

Trade-off topics

- Intercepting calls to a memory allocator
- Collection of basic data and counting
- Enhancements to profiling

Using a custom memory allocator

- Why don’t we just fork off jemalloc or mimalloc or TCMalloc and change it the way we like it?
  - Now we have two problems.
  - A new problem is nontrivial—make everything work with a custom allocator.
- Well, why don’t you just have a custom version of libmalloc?
  - Legal issues aside …
  - The reference source code is a version from some point in past.
  - User mode allocators and OS tend to have ”an understanding” and there’s no way for us to account for undocumented behavior.
  - Any OS update can break everything.

Overriding the allocator: code injection

- Bad idea in general (just another exploit primitive).
- Linux: `LD_PRELOAD` + `dlsym(RTLD_NEXT, …)`
- macOS: `DYLD_INSERT_LIBRARIES` or `DYLD_INTERPOSE`
  - Requires System Integrity Protection (SIP) disabled.
  - New categories of problems on devices (e.g., iOS).
- Windows
  - Many ways
  - Closest to previous examples is Detours.

`ld` and `--wrap` flag (1)

- No `ld` support on macOS.
- Does not wrap already compiled code (e.g., a system dynamic library).

```
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size);
    // Need to avoid infinite recursion.
    nomalloc_printf("malloc(%zu) = %p\n",
                    size, ptr);
    return ptr;
}


clang ./foo.c -Wl,--wrap,malloc -Wl,--wrap,free
```

`ld` and `--wrap` flag (2)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char *s = strdup("This allocation is not tracked ;-(");
    puts(s);

    return EXIT_SUCCESS;
}
```

The “constructor” attribute

- Used by custom memory allocators (e.g., jemalloc, mimalloc).
- Several weird corner cases. Look at source code of various allocators for detailed explanations.
- “What if everyone did that?”

```
__attribute__((constructor))
static void
init_something(void)
{
    . . .
}
```

Built-in interceptors (1)

- IMHO the optimal method.
- Manipulating the hooks is not thread-safe.
- The GNU C library used to have a built-in mechanism to replace a built-in malloc implementation.
  - Removed since glibc version 2.34 (August 2021).

Built-in interceptors (2)

- Android supports malloc hooks that are "only available in Android P and newer versions of the OS."

```
void* new_malloc_hook(size_t bytes, const void* arg) {
  // Do whatever you need to do here ...
  return orig_malloc_hook(bytes, arg);
}

auto orig_malloc_hook = __malloc_hook;
__malloc_hook = new_malloc_hook;
```

Built-in interceptors (3)

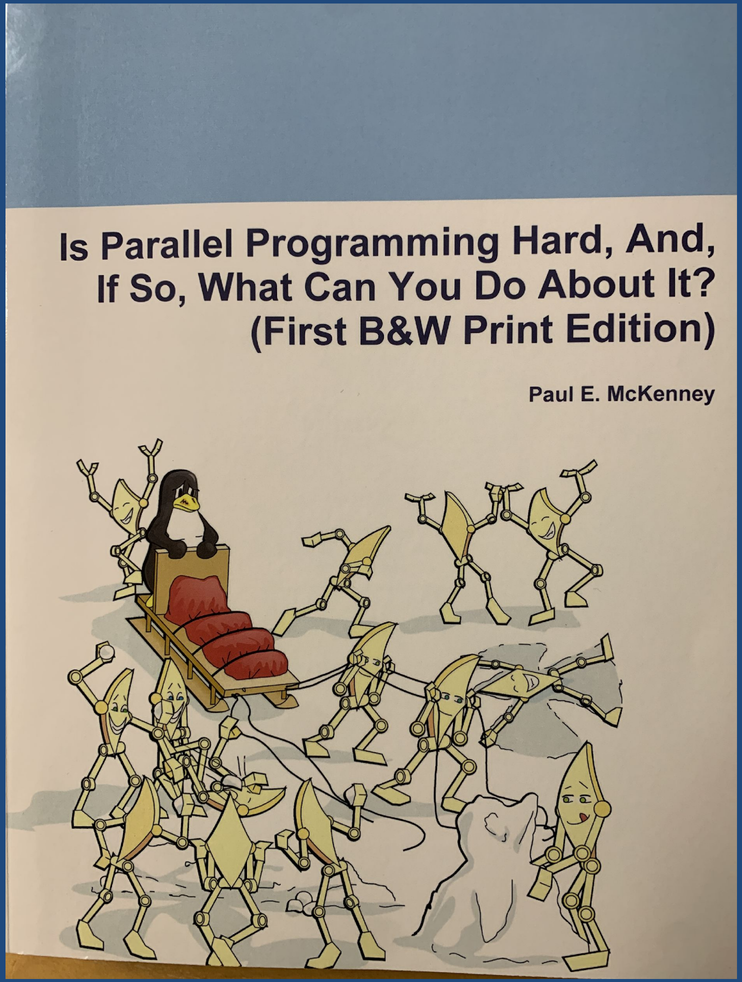
- For macOS:
  - `malloc_logger`
  - `__syscall_logger`
- [libmalloc source code] is the documentation.
- NB! Manipulating the hooks is not thread-safe.

```
static void *

_malloc_zone_malloc {

  . . .


  if (os_unlikely(malloc_logger)) {
    malloc_logger(MALLOC_LOG_TYPE_ALLOCATE | ... ,
                  (uintptr_t)zone,
                  (uintptr_t)size,
                  0, (uintptr_t)ptr, 0);
  }


  . . .
}
```

Temptation

- We’ll design and implement our own intercept mechanism.
- Why? So, that we can intercept everything everywhere.
- It’s like Poor Man’s Rootkit.
- Why? Because it’s cool.

Counting is hard …



5 Counting
- 5.1 Why Isn't Concurrent Counting Trivial?
- 5.2 Statistical Counters
  - 5.2.1 Design
  - 5.2.2 Array-Based Implementation
  - 5.2.3 Per-Thread-Variable-Based Implementation
  - 5.2.4 Eventually Consistent Implementation
  - 5.2.5 Discussion
- 5.3 Approximate Limit Counters
  - 5.3.1 Design
  - 5.3.2 Simple Limit Counter Implementation
  - 5.3.3 Simple Limit Counter Discussion
  - 5.3.4 Approximate Limit Counter Implementation
  - 5.3.5 Approximate Limit Counter Discussion
- 5.4 Exact Limit Counters
- 5.5 Parallel Counting Discussion

Start with simple solutions

- Global synchronization primitive (e.g., a mutex)
- RWL (reader-writer lock)
  - Our scenario: a lots of writing, occasional reading.
- Spinlocks—meh …
- `std::atomic<T>`
  - First glimpse of hope.
- Temptation:
  - We’ll implement our own RCU (read-copy-update) and it’s going to be great!
- Eventual (obvious?) focus: TLS (thread-local storage).

TLS (thread-local storage)

- Not a magical solution.
- Something somewhere needs to implement TLS.
- A typical implementation is lazy and uses `malloc()` to maintain TLS entries.
- Guess, what happens when you call `malloc()` during the `malloc()` intercept?
  - Reentrancy problems come calling.
- General problem: you should be very conscious of what you execute during the intercept.
  - What is the contract for locks, signals, triggering allocations?

When malloc() intercept calls malloc() …

```
...
116 libdyld.dylib             tlv_get_addr + 296
117 a.out                     _ZL24libmalloc_intercept_funcjmmmmj + 48
118 libsystem_malloc.dylib    _malloc_zone_malloc + 249
119 dyld                      _ZN5dyld412RuntimeState16_instantiateTLVsEm + 175
120 libdyld.dylib             tlv_get_addr + 296
121 a.out                     _ZL24libmalloc_intercept_funcjmmmmj + 48
122 libsystem_malloc.dylib    _malloc_zone_malloc + 249
123 dyld                      _ZN5dyld412RuntimeState16_instantiateTLVsEm + 175
124 libdyld.dylib             tlv_get_addr + 296
125 a.out                     _ZL24libmalloc_intercept_funcjmmmmj + 48
126 libsystem_malloc.dylib    _malloc_zone_malloc + 249
127 dyld                      _ZN5dyld412RuntimeState16_instantiateTLVsEm + 175
```

Temptations

- We will write our own TLS implementation.
- Why? Because it's cool.
  - Also, jemalloc team has worked on something similar.
- Track each time a new thread is created and destroyed. Do some trickery to "pre-initialize" TLS.
  - Analogous to `for_each_thread()`.
  - A non-trivial problem with a highly implementation-dependent solution.

Hacks to work around the TLS initialization

- Make a design decision that we'll track only first N threads.
  - Use (supposed) guarantees from C++ memory model and manually tune memory ordering.
  - Some basic RCU manipulation to keep track of indices.
- Use a global data structure to manage the state.
  - `mach_port_t mach_tid = pthread_mach_thread_np(pthread_self());`
  - `int idx = ƒ(mach_tid); // O(1) lookup`
  - `// Play around with a[idx];`
- The tracking data structure serves mainly reads, writes only when a new thread is created.
- Track a limited number of variables, pack them, try to avoid CPU cache ping-pong as much as possible.

Pseudo-algorithm

```
int idx = f(mach_tid);                    // O(1) magic.

if (a[idx].state == UNINITIALIZED) { // (R) 1st on this thread.
  a[idx].state = IGNORE;                  // (W) Ignore next calls.
} else if (a[idx].state == IGNORE) { // (R)
  return;                                 // Reentrant call – bail out.
}

do_something_with_tls_variables();   // Causes reentrancy.

if (a[idx].state == IGNORE) {          // (R)
   a[idx].state = INITIALIZED;         // (W)TLS access:initialized.
}
```

Temptation

- We'll implement a custom versions of typical dynamic data structures (e.g., a hash table) that use a lower-level allocation primitives to avoid the `malloc()` dependency.
  - Why? Because we can.
- We'll implement a cool way to "avoid" locks by using lock-free data structures.
  - Why? Because we've read all those papers and books about lock-free programming, and it sounds really cool.

Problems with the implementation

- Tooling from Apple (e.g., Xcode Instruments) uses the same mechanism to intercept the allocations can't have both run at the same time.
- Sanitizers such as AddressSanitizer (aka ASan) will have their own malloc implementations.
- Custom allocators that manage their own arenas/heaps/zones will avoid the system libraries. For example, mimalloc in standard configuration will use only `mmap()` and `munmap()`.
- Restricted to libmalloc only. User can always call lower-level APIs such as `mach_vm_allocate()`. Those can be intercepted as well.
- Tracking the **real** allocated size (e.g., result of `malloc_size()/malloc_usable_size()`) is costly.

Sample API (based on libmalloc definitions)

```
typedef void(malloc_logger_t)
  (uint32_t type, uintptr_t arg1,
   uintptr_t arg2, uintptr_t arg3,
   uintptr_t result, uint32_t num_hot_frames_to_skip);

extern malloc_logger_t *malloc_logger;

typedef struct _malloc_stats_t {
...
} malloc_stats_t, *pmalloc_stats_t;
```

Temptation

- We'll use as much futures, lambdas, and promises as we can to implement the API.
  - C++ 11/14/17/20/23
- Why? Because C++ now supports all kinds of cool things. We want to use the latest standard because it's there.

Sample API (usage patterns)

```
int start_malloc_trace();
int stop_malloc_trace();

int reset_thread_malloc_stats();
int reset_global_malloc_stats();

int get_global_malloc_stats(malloc_stats_t *global_stats);
int get_thread_malloc_stats(malloc_stats_t *thread_stats);
```

- Custom classes that use RAII pattern on top of it to make the usage easy.
- Hide everything from the consumer.

More complex scenarios

- If you can't keep everything in memory, then you need to use some form of storage.
- Storage (typically a disk) means opening multiple cans of worms.
  - Shared (circular) queues in the memory.
  - Concurrent access by reader and writer threads.
  - Asynchronous and synchronous I/O decisions.
  - Managing the data store (e.g., cleanup, compaction, limits).
  - Data corruption.
  - Packaging, compressing, transmitting, and decompressing the data.

Final thoughts

- It has been some time since we've worked on this problem—take everything that was said with a grain of salt.
- Ideally, the allocator should keep track of statistics
  - Both jemalloc and TCMalloc expose some.
  - Default allocators don't share much.
    - **glibc:** struct mallinfo mallinfo(void);
    - **glibc**: struct mallinfo2 mallinfo2(void);
- Using custom memory allocators causes an "intercept race condition."
- Are TLS models in LLVM something that can be used?

Acknowledgements

- The only reason I am here is because of the kindness of LLVM Foundation.

THANK YOU - ENGINEERS NEED YOUR HELP!

Interested? Intrigued?
Disagree? Collaborate?
Have a beer? Go for a trail run?

- gunnarku.github.io
- g.kudrjavets@rug.nl

Everything will be fine

"A systems programmer will know what to do when society breaks down, because the systems programmer already lives in a world without law."

— James Mickens, The Night Watch, 2013

References

- Duffy, J., & Sutter, H. (2008). *Concurrent Programming on Windows*. Boston, MA: Addison-Wesley Educational.
- Herlihy, M., & Shavit, N., (2020). *The Art of Multiprocessor Programming (2nd ed.).* Burlington, MA, USA: Morgan Kaufmann Publishers.
- Kerrisk, M. (2010). *The Linux Programming Interface*. San Francisco, CA, USA: No Starch Press.
- Levin, J. (2017). **OS internals: User space*. White Plains, NY: USA: Technologeeks Press.
- McKenney, P. E., (2014). *Is Parallel Programming Hard, And, If So, What Can You Do About It?* Self-published.
- Singh, A. (2006). *Mac OS X Internals*. Boston, MA: Addison-Wesley Educational.
- Williams, A. (2019). *C++ Concurrency in Action (2nd ed.).* New York, NY, USA: Manning Publications.