Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Eliminating Small, Transient Memory Allocations

Dário Tavares Antunes
B. A. (Mod.) Computer Science
Final Year Project May 2018
Supervisor: Dr. David Abrahamson

School of Computer Science and Statistics
O'Reilly Institute, Trinity College, Dublin 2, Ireland

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

Dário Tavares Antunes, January 1, 1970

Permission To Lend

I agree that the Library and other agents of the College may lend or copy this report upon request.

Dário Tavares Antunes, January 1, 1970

Abstract

This is where I would put the abstract, IF I HAD ONE

Acknowledgements

This is where I would put my acknowledgements, IF I HAD ANY

Contents

Chapter 1

Introduction

> 640K ought to be enough for anybody.
>
> Not Bill Gates

Despite the often misattributed epigraph above often being used to mock past beliefs that some amount of memory should be enough for any reasonable purposes, the mentality behind it is still pervasive.

With the broad availability of large amounts of computational power, memory and storage, conservation or efficient use of the same is often overlooked in programming. This is largely perpetuated by the (often valid) view that programmer time is more valuable than the benefits that more efficient but more complex code brings.

However, there remain situations where these benefits are in fact worth the effort required. One of these such cases is in code intended to be deployed in embedded or mobile devices, where resources are limited and preservation of power is essential.
A blog post [Ste17] by Daniel Stenberg, original author of the *curl* command line tool and ubiquitous URL data transfer tool, is a retrospective on an attempt to reduce unnecessary heap allocations.

Inspired by that post, the aim of this project is to produce a tool to identify cases where similar changes could be made in order to potentially reduce a program's energy and processing power footprint, and at the same time improve its performance.

1.1 Report Structure

The report is structured as follows:

- Chapter 2 provides background on the project, including further information on the changes to *cURL* that inspired this project, as well as laying out the objectives of the project

- Chapter 3 describes the goals of the plugin developed, the platform it built upon, difficulties encountered in development, and the final state of the plugin
- Chapter 4 covers three case studies, two written intentionally to trigger certain behaviours to maximise the optimisation's effect, and one which simply involves isolating Stenberg's changes and testing their impact
- Chapter 5 examines the results and outcomes of the case studies and state of the plugin, as well as including a short discussion on potential benefits of future work in this area
- Chapter 6 describes the state of the art in related areas to the project
- Chapter 7 describes some areas with potential for future work

Chapter 2

Background and Objectives

This chapter will contain:

2.1 Background on the Patch

cURL's first dated change was introduced in April of 1998 [Con98], with three versions already having been released before that. When introduced, guidelines for contributors were loose and didn't particularly discourage varying programming styles or adherence to existing styles in the codebase [Con99]. In the 20 year interim, stricter guidelines have been introduced, all changes require tests and must be sufficiently atomic and so on [Con17].
However, over 150,000 lines of C have been added in that period of time and under potentially weaker requirements. As a result, there are plenty of places where improvements can be made.

In particular, Stenberg's post discusses two allocation related changes [Ste17]. The first involves rewriting some generic linked list functions in order to remove all allocation from them, while the second involves rewriting a polling function which takes a copy of its input to copy said input into a stack-allocated buffer in a common case, rather than using `malloc` every time.

2.1.1 Linked List Changes

The original linked list implementation incurred a `malloc` on every insertion and a `free` on every deletion, as the data and the linked list node were two separately allocated objects. First the data struct would be allocated and initialised, then passed to the linked list functions, which would then allocate a linked list node struct and point it at the data struct before continuing on to perform the requested operation.

The change involved rewriting any data structs to also contain a linked list node struct and changing the generic linked list functions to take both a pointer to the data struct and a pointer to the linked list struct (which would just point

at the struct contained in the data struct, while allowing the functions to remain generic). This has two beneficial results:

- Linked list functions can't fail due to memory constraints any more, simplifying logic that uses them
- Less allocations are performed, as only one allocation is performed per node rather than two

According to Stenberg in his blog post, these changes led to a modest reduction in the number of allocations in a simple benchmark (from 115 allocations to 80, or a 26% reduction) [Ste17]. Stenberg notes that these changes are effectively free, and improve the code quality.

2.1.2 Polling Function Changes

The polling function in question is `curl_multi_wait`. The function takes as input a list of file descriptors[1], polls each one and returns with an error code (indicating whether the descriptors were polled successfully or if there was some issue).
For the purposes of polling, *cURL*'s internal abstraction is accepted alongside regular file descriptors. In order to make their handling simpler, a block of memory is allocated with a plain `malloc` where all file descriptors are copied to for polling.

The expectation is that `curl_multi_wait` will be used in conjuction with other functions for bulk operations on sets of file descriptors in a polling loop. Due to internal constraints on timeouts, this means that `curl_multi_wait` could be called as often as 1000 times per second, each time potentially calling `malloc`. Removing this `malloc` should lead to a significant reduction in the number of allocations made.

The change made here was simple, and the one of interest for this project. In the common case (as claimed by Stenberg without mentioning how its commonness was determined), `curl_multi_wait` was changed over to avoid the malloc and instead use a stack allocated block of memory when few file descriptors were passed to it.
There was a very significant claimed decrease in the number of allocations as a result of this change in a simple benchmark (from 33,961 to 129, or a reduction of 99.62%).

2.1.3 Results of the Changes

The version of the tool built with these changes was then compared in a fully local benchmark (to avoid any impact of network connectivity or other external factors) to the previous release. Stenberg reports it performed 30% faster, transferring 2900 MB/sec vs the previous version's 2200 MB/sec.

[1] A file descriptor is part of the POSIX API, providing a uniform interface to similar but distinct interfaces such as files, hardware devices, network sockets and so on. *cURL* further abstracts the concept for added portability. The specifics are not important here.

However, this comparison attributes all performance and allocation differences to these two commits, despite there having been 231 commits in total between the two versions. Stenberg highlights this, but adds a caveat that none of them spring to mind as having an impact on the number of allocations or significant performance changes.

2.2 Objectives of this Project

There are two main objectives for this project.

1. Produce a tool that can detect sites where there is potential for the patch to be performed
2. Determining the performance impact the patch can have

2.2.1 Tool to Detect Potential Patch Sites

The general pattern of sites where this patch can be applied appears something like the below

```
1 int func(size_t alloc_size) {
2   void* alloced = malloc(alloc_size);
3   int result = // do things with alloced
4   free(alloced);
5   return result;
6 }
```

where the `malloc` and `free` on lines 2 and 4 could instead be replaced with stack allocation[2], avoiding both of those calls and indeed completely avoiding any risk of a memory leak[3].

The concept is simple: some amount of memory is allocated, used for a short amount of time, then `free`d. In a small example, the pattern is obvious and easy to detect, or even to not introduce in the first place. However, as seen in the real world *cURL* example, these patterns are introduced, either by mistake or for simplicity.

There are also further considerations to be taken before replacing a heap allocation with a stack allocation, and even more considerations if it's to be replaced with static allocation. A non-comprehensive list follows, where some items result in undefined behaviour[4]

- Stack overflow can be caused by stack allocation of a large amount of data, resulting in UB

[2] The details of how stack allocation would be achieved in this situation is explored further later, the details are unimportant at this point

[3] A memory leak refers to a dynamic allocation (using the `malloc` family or similar) which is never `free`d and so consumes memory until the program exits, even if it's no longer being used

[4] Undefined behaviour in C is the result of any operation which has no defined semantics, and its outcome may vary from implementation to implementation or even run to run. To the compiler, it is equivalent to $\perp$, and so it may generate any code if it can detect UB

- A pointer to the data escaping its scope would result in a dangling pointer, resulting in UB
- The variable may be assigned at various different points, complicating stack allocation (depending on the method used)
- If static allocation is used, it must be guaranteed that the function can only be executed in one site at a time to avoid multiple sites overwriting each other's data

The tool should take as many of these cases into consideration as possible, to avoid suggesting sites for the patch to be applied where it would cause errors.

Development of the tool is discussed in depth in chapter 3.

2.2.2 Determining the Patch's Performance Impact

First, the maximum expected performance impact should be found, to set an expectation of what the best case would be. To that end, two bespoke benchmarks were written: one to attempt to trigger certain slow behaviours in the allocator that can then be avoided by stack allocating instead; another to simulate a simple but realistic benchmark to test the results of the patch in isolation.

Next, in order to determine the performance change in a real world situation, the *cURL* patch itself was tested in complete isolation from the other commits to determine how much of the performance difference was a result of the allocation changes.

The benchmarks are discussed in depth in chapter 4.

Chapter 3

Implementation

This chapter will contain descriptions of:

3.1 Goals of the Plugin

What are the initial goals of the plugin, including those that weren't achieved? How do these goals directly relate to the patch?

3.2 The Frama-C Platform

What is Frama-C? Why choose Frama-C (over bespoke solutions or other platforms)? Background on Frama-C, its goals.

3.3 Difficulties Encountered

What difficulties were encountered (new language, installation, documentation, usefulness of results from other plugins [no location data in Base], any future issues)

3.4 Final State of the Plugin

What state is the plugin currently in? How well has it achieved its goals? Refer to future work section.

Chapter 4

Case Studies

This chapter will include, for each chosen case study:

4.1 Case Study 1

4.1.1 Program Reasoning

Why choose Case Study 1 to attempt to apply the patch?

4.1.2 Predictions

What results are expected as a result of the patch being applied to Case Study 1?

4.1.3 Patch Code

Either a listing of the patch inlined here, referred to the appendix, or referred to the attached source code or GH. Also needs a commitish for the version of the code to apply the patch to.

4.1.4 Results

What were the results before and after the patch? (number of mallocs, performance, speed, power usage if measurable)

4.1.5 Comparison to Predictions

How did the predictions line up? Better/worse?

4.1.6 Hypothesis

Why were the results what they were?

Chapter 5

Conclusion

This chapter will include high level summaries and conclusions on:

5.1 Results

Did it match the predictions, if not, hypothesize why not

5.2 State of the Plugin

What state is the plugin left in, is there much work to be done, what would be done with more time, what benefits would those changes have/what are the priorities

5.3 Benefit of Further Work

Would further work in this space (not specifically the plugin) be beneficial? If so, why/what should be done first?

Chapter 6

State of the Art

6.1 State of the Art

6.2 Similarity of the Patch to Concepts in Generational Garbage Collection

Background based on similarity to generational GC assumptions (Appel, Shao)

6.3 Predictions on Results of Generalisation of the Patch

Predictions on the results of a generalised application of the patch (vs proebsting's law for compilers)

Chapter 7

Future Work

> A program is never less than 90% complete, and never more than 95% complete.
>
> Terry Baker

Any non-trivial work is never complete. To that end, listed below are some ideas for potential improvements on the *forgetful* plugin or related works on the same principle.

7.1 Detecting Arbitrary Memory Allocations

The current implementation only finds allocations based on uses of malloc and free. Other ways to allocate memory exist (`calloc`, `realloc`, `alloca`, direct uses of `mmap` and `sbrk`), and platforms that stand to gain the most from this optimisation may have their own implementations.

An extension to this work could involve allowing an arbitrary list of functions declared to allocate or deallocate memory, potentially with fully annotated files specifying their behaviour so that `frama-c` can be used to its full potential (particularly for value analysis, which relies on these specifications).

Alternatively, if there is a willingness to assume a unix-like platform, the depth of analysis could be extended to attempt to automatically determine which functions might allocate memory by searching for `mmap` or `sbrk` calls and propagating annotations indicating functions that directly or indirectly allocate memory.

The approach propagating allocation information already exists in some form in Facebook's Infer [Fac13] static analyser, so future work could also involve extending that platform instead.

7.2 Automatically Performing Fixes

Ideally, fixes would be automatically generated and patched into the code at compile time, avoiding added complexity from the programmer's point of view while still taking advantage of the performance and memory benefits.

Potential intermediate steps toward that goal could involve generation of patches that could be applied to code before compilation, introducing the optimisation. Fortunately, `frama-c` already has code generation capabilities which could be taken advantage of for this purpose.

7.3 Studying Other Architectures

For the sake of practicality and convenience, the analysis presented was only performed on a single x86 machine. To be certain whether these results apply more generally, the analysis should also be performed on other architectures (for example: ARM, x64, embedded systems without address translation/paging, SPARC).

Bibliography

[Con98] cURL Contributors. *cURL ChangeLog*. 1998. URL: https://github.com/curl/curl/commit/ae1912cb0d494b48d514d937826c9fe83ec96c4d#diff-e4eb329834da3d36278b1b7d943b3bc9R1379 (visited on 04/13/2018).

[Con99] cURL Contributors. *cURL Contributing Guidelines*. 1999. URL: https://github.com/curl/curl/blob/ae1912cb0d494b48d514d937826c9fe83ec96c4d/CONTRIBUTE (visited on 04/16/2018).

[Fac13] Facebook. *Facebook Infer*. 2013. URL: http://fbinfer.com/ (visited on 03/01/2018).

[Con17] cURL Contributors. *cURL Contributing Guidelines*. 2017. URL: https://github.com/curl/curl/blob/36f0f47887563b2e016554dc0b8747cef39f746f/docs/CONTRIBUTE.md (visited on 04/16/2018).

[Ste17] Daniel Stenberg. *Fewer mallocs in curl*. 2017. URL: https://daniel.haxx.se/blog/2017/04/22/fewer-mallocs-in-curl/ (visited on 02/01/2018).