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.

There are cases where such benefits remain essential, such as in code intended to be deployed in embedded or mobile devices, where resources are limited and preservation of power is paramount. It can also provide benefits to real-time systems in which the potential delay caused by a worst-case allocation is unacceptable. An added benefit presents itself in the case of a ubiquitous library, where even a small improvement can lead to large energy savings on the aggregate once the large number of users are accounted for.
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 plug-in developed, the platform it built upon, difficulties encountered in development, and the final state of the plug-in
- 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 plug-in
- 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 describe the background of the patch, discussing what leads to its introduction to a codebase with a specific focus on *cURL*. The project goals are then defined in the context of the patch to be detected and benchmarked.

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 encourage 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. The file descriptors are all copied to a block of memory, allocated with a plain `malloc`, to make polling from the two distinct sources simpler.

The expectation is that `curl_multi_wait` will be used in conjunction 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 to avoid the `malloc`, instead using a stack allocated block of memory when few file descriptors were passed to it.
Stenberg claims that in a simple benchmark this change resulted in a 99.62% decrease (33,961 to 129) in the number of `malloc` allocations.

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 exter-

[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.

nal factors) to the previous release. Stenberg reports it performed 30% faster, transferring 2900 MB/sec vs the previous version's 2200 MB/sec.

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

The project has two main objectives.

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

2.2.1 Tool to Detect Potential Patch Sites

The general pattern of sites where this patch can be applied appears something like the code listing 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] (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

- Stack overflow can be caused by stack allocation of a large amount of data, resulting in 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 applying the patch at sites where it would cause errors.

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

2.2.2 Determining the Patch's Impact on Performance

First, the maximum expected performance impact should be found, to set an expectation of what the best case result 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

Two initial approaches to create the tool were considered: hooking directly into the compiler to detect the pattern and automatically patch it (when enabled, and when the pattern is detected with sufficiently high confidence); or creating a plug-in for an existing static analysis platform which could be manually run on existing codebases to detect the pattern.

The time needed to become familiarised with the extensive and complex codebase of a production compiler (which may not even provide sophisticated value analysis) would lead to a highly reduced chance that the project would be complete within the time allowed.
Therefeore the final decision was, largely for reasons of pragmatism and convenience, to follow the second approach and create the Forgetful plugin.

3.1 Goals of the Plug-in

There were a small set of goals for the plug-in to achieve, both functional and non-functional.

3.1.1 Non-Functional Goals

The non-functional goals are as follows:

- There should be little to no modification of any existing code required to use the plug-in to a satisfactory degree
- There should be a minimal number of false positives wherein the plug-in suggests a site at which the patch cannot be applied
- Interaction with the plug-in should match the normal mode of interaction for the platform it builds on

These goals should ensure that the barrier to entry to using the plug-in is as low as possible, as it can be used directly on existing code, even if the static

analysis platform itself has never been used on that code. Additionally, avoiding false positives makes it more likely that action will be taken on the plug-in's results by minimising the amount of data users have to trawl through [Vil17]. Lastly, ensuring all interaction with the plug-in matches what's expected of its platform makes its adoption in systems already using the static analysis platform easier.

3.1.2 Functional Goals

The functional goals are as follows:

- When a site where the patch can be applied is found, the user should be notified
- Where possible, a diff patch[1] should be produced to apply the patch easily

On the first point, the user clearly needs to be notified, as there's no point to detecting an issue and not communicating it to the user. The exact form of the notification isn't important, but should provide as much information as possible without overwhelming the user, allowing them to make a informed decision about what action to take.

The diff patch is more complicated, but would be extremely useful. If the plug-in could guarantee that a certain site could be patched safely before producing a diff patch, it could be added into a pre-compilation step to rewrite the pattern silently. This would allow the source code that users work on to remain simple and as they wrote it while gaining any performance benefit from the patch.

3.2 Static Analysis Platform

There are a number of static analysis tools built for C over the years, of which a small number were chosen based on apparent state of maintenance and popularity (as a proxy for likelihood to be well supported and modern). The shortlist from which the eventual target platform was chosen consisted of *clang-analyzer* [LLV07], *Frama-C* [Fra08f], and *Infer* [Fac13].

clang-analyzer is written in C++, matching the *clang* codebase it originates from and resides in. *Frama-C* and *Infer* are both written in OCaml, though while *Frama-C* builds up its own AST[2], *Infer* hooks into *clang-analyzer*.

The two tools that were not chosen are discussed in further depth in Chapter 6 in comparison to *Frama-C* in a retrospective manner.

[1] A diff patch is an encoding of a set of changes that can be automatically applied with a standard tool to a file to effect a change

[2] An Abstract Syntax Tree (AST) is a tree-based representation of a program, with each node representing a construct appearing in the source code

3.3 The *Frama-C* Platform

The static analysis platform chosen was *Frama-C*. *Frama-C* has an emphasis on correctness, providing its own language for functional specifications which can be provided alongside the code. While this is of no particular interest to this project due to the first functional goal, it assists in reducing false positives as a result of its conservative approach and care around sites of potential undefined behaviour [Fra08g]. Additionally, that specification language is used by the platform to provide properties of standard library functions such as `malloc` and `free`, which is essential to the project's analysis.

However, and of more interest to the project, it also has a plug-in architecture, which makes it easy to extend and build on. In particular, it enables plug-ins to interact, which allows new plug-ins to use functionality exposed by existing plug-ins thereby reducing the workload required within the plug-in itself. This was the primary factor in the choice of *Frama-C* over the other two platforms [Fra08c].

3.3.1 Source Code Processing

Frama-C produces an AST which plug-ins can then operate on. The version of the code exposed to plug-ins is normalised by Frama-C, which prevents duplication of efforts in handling unusual edge cases enabled by C's permissive design. As an example, consider the following C functions (which are intentionally contrived)

```
int fc(int a) {
  return a + 1;
}

int main(void) {
  int i = 1;
  int* point = malloc(sizeof(i));
  int** ppoint = malloc(sizeof(point) * fc(1));

  return 2 * i;
}
```

This is normalised into something like the following by *Frama-C*

```
int fc(int a)
{
  int __retres;
  __retres = a + 1;
  return __retres;
}

int main(void)
{
  int __retres;
  int **tmp_1;
  int tmp_0;
  int i = 1;
  int *point = malloc(sizeof(i));
```

```
15   tmp_0 = fc(1);
16   tmp_1 = (int **)malloc(sizeof(point) * (unsigned int)tmp_0);
17   int **ppoint = tmp_1;
18   __retres = 2 * i;
19   return __retres;
20 }
```

We note in particular that rewrites are performed in order to avoid multiple operations occurring on a single line, such as splitting out the evaluation of return values and their actual return, or the evaluation of expressions involving function calls and the actual function call. This prevents an arbitrarily complex AST from being constructed.

The AST as provided to plug-ins to traverse is also annotated. It can be annotated in the source code itself, using *Frama-C*'s ACSL[3] to add specifications [Fra08a], or annotations can be added by other plug-ins as they discover properties of the code [Fra08d].

The root of the AST is a representation of the file being processed, which contains a collection of globals, of which we're only interested in functions. Other globals include declarations of variables, types, structs, unions, and enums among others.

Within a function node we're interested in its statement list, which contains statements of various kinds, such as a plain instruction with no control flow, which can include a variable declaration and assignment, or a reassignment of an existing variable. These are the exact subsets of statements in which a `malloc` can occur after normalisation of the AST, including the unusual case of a `malloc` that's not assigned to anything.

Frama-C alone doesn't provide any sort of value or escape analysis, instead leaving this to be provided by plug-ins. The primary plug-in providing these features is called *Evolved Value Analysis* (*EVA*). Note that this distinction is largely symbolic, as *EVA* is statically connected to the *Frama-C* kernel, unlike regular plug-ins.

3.3.2 The *Evolved Value Analysis* Plug-in

EVA provides, at any given point in the AST, a set or interval describing values possible at a given point. Values can be requested for expressions or variables with respect to a given statement, and they can be evaluated either before or after execution of that statement. *EVA* also performs semantic constant folding, allowing it to be used even on code including loops [Fra08b]. This will be needed in order to track allocations throughout the program.

Values can be described as a discrete set of values, as an interval, or as an interval skipping regular values. When *EVA* determines that one of the representations is becoming too large, it can degenerate the value to a broader description that contains all of the original values. As an example, take the following:

[3] ANSI/ISO C Specification Language, used to formally define function contracts

Variable	Values	Variable	Values
`morePrimes[0]`	{2}	`morePrimes[4]`	{11}
`morePrimes[1]`	{3}	`morePrimes[5]`	{13}
`morePrimes[2]`	{5}	`morePrimes[6]`	{17}
`morePrimes[3]`	{7}	`morePrimes[7]`	{19}
`randVal`	[0..32767]	`index`	{8}
`randPrime`	{2; 3; 5; 7; 11; 13; 17; 19}		

Table 3.1: Values provided by EVA with `NUM_PRIMES` set to 8 and sufficient semantic constant folding to fully evaluate the loop

```
1  int main(void) {
2    srand(time(NULL));
3    int randVal = rand();
4    int randPrime = 2;
5    int morePrimes[NUM_PRIMES];
6    int index;
7    int current = 1;
8
9    for (index = 0; index < NUM_PRIMES; current++) {
10     if (isPrime(current)) {
11       if (rand() % 2) {
12         randPrime = current * 4;
13       }
14       morePrimes[index] = current;
15       ++index;
16     }
17   }
18 }
```

Assuming that the level of semantic constant folding *Frama-C* is permitted to do is high enough to fully evaluate the loop and that `NUM_PRIMES` is set to 8, *EVA* produces the values in Table 3.1 after the loop.

In particular, note that:

- each item in the array `morePrimes` is tracked separately by *EVA*
- `randPrime` can take on any of the prime values
- `randVal` can take on any values between 0 and *Frama-C*'s `RAND_MAX`
- variables that can only take on a single value are considered to have a value which is a singleton set.

Next, we consider the values reported if the semantic constant folding allowed is set too low to evaluate anything past the first prime, and so the values are now reported as shown in Table 3.2.

This time we note that EVA can no longer determine whether the loop ever terminates and cannot determine the values for all indices of the `morePrimes` array, nor if they're ever initialised (due to the possibility of integer overflow in `current` without index being incremented sufficient times to exit the loop). As expected, `randPrime`'s possible values also cannot be determined, as it depends

Variable	Values	Variable	Values
`morePrimes[0]`	{2}	`index`	{8}
`morePrimes[1..7]`	[3..2147483647] or UNINITIALIZED	`randVal`	[0..32767]
`randPrime`	[2..2147483647]		

Table 3.2: Values provided by EVA with `NUM_PRIMES` set to 8 and insufficient semantic constant folding to fully evaluate the loop

on full evaluation of all values in `morePrimes`. However, the other variables do not depend on the loop, so they can be correctly evaluated regardless.

Next, increasing `NUM_PRIMES` to 9 causes *EVA* to decide `randPrime` has too many values, so it reduces its precision from the values shown in Table 3.1 to [2..23] which still contains all the correct values with as much precision as possible without storing the individual values.

Additionally, changing line 12 so that `randPrime` is assigned `current * 4` causes *EVA* to degenerate its precision again to [2..76]0%2, which is the most complicated value type *EVA* can produce, and indicates that values start at 2 with an offset of 0 and every second value is potentially valid. This includes all the valid values ({8; 12; 20; 28; 44; 52; 68; 76}), but is less precise than an alternative interval of [8..76]0%4. It's not clear why *EVA* choose one instead of the other.

While this covers all simple values such as integers, floats, and even structs (which function similar to an array, where each member is separately displayed), it doesn't cover pointers. There are two kinds of pointer which are represented identically. The first is a pointer to an existing variable such as `&randPrime`, while the second is a pointer created by a call to a function like `malloc`. Both types are represented as {{ `&varname` }}, where `varname` is either the name of the variable pointed to, or something of the form {{`&__malloc_main_l33` }} in the case of `malloc`, where a unique variable name is generated, representing a point in heap memory which *EVA* calls a Base.

Bases can be collected in sets, same as regular values, but cannot form an interval. There is also a special pointer, `NULL`, representing exactly that and marked as a potential return value by *Frama-C*'s internal version of `malloc`, although that can be disabled by one of *EVA*'s options [Fra08e]. Clearly, bases are of particular interest for the project.

3.4 Development of the *Forgetful* Plug-in

A plug-in development guide is provided to aid new developers in the *Frama-C* environment to create their own plug-ins [Fra08d]. The guide outlines some common use-cases, providing some code samples and best practices. Some of these were used in the creation of the *Forgetful* plug-in.

3.4.1 Visitor Pattern

For any plug-in that doesn't require direct access to the AST for any particular reason, the development guide recommends usage of one of the provided visitor classes built-in to *Frama-C*.

These are classes implementing the visitor design pattern, intended for usage by developers who can extend and override only the specific methods they're interested in. The design pattern itself is described as

> Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. [RHR94]

The benefit of this pattern is that it allows the easy addition of new operations on the AST, with the downside being that it's difficult to add new types of node to the AST, but given the nature of the AST new types of nodes are rare.

Concretely, for the development of the plug-in, this means that it can simply extend the in-place visitor (since the plug-in will not modify the AST itself, otherwise it would have to use the copy visitor to avoid corrupting information already attached to the AST [Fra08d]) and override the functions for visiting individual statements and for visiting the function declaration node.
This will allow the plug-in to track which function it's currently in for scoping purposes, and to inspect the contents of statements in order to determine if they contain a `malloc` or `free`.

Frama-C's documentation is limited, with many types having no documentation or referring the reader to either the plug-in development guide or the user guide with no indication of what section within the guides are relevant. As such, determining what purpose certain nodes in the AST served had to be determined through trial and error. For example, the `Block` node represents a block (such as a loop body) and so contains a list of statements, but the visitor doesn't have to traverse those statements as they're actually duplicated.
Trial and error was also the method used to determine which nodes `malloc` and `free` could appear in after normalisation of the AST.

3.4.2 Allocation Tracking

Allocation tracking is performed only within any given function so as to ensure any allocations whose free site is found are short-lived, and specifically inter-procedural. This is non-essential, but is the easiest case of the pattern to replace with stack allocation.

To actually track allocations, a hashtable mapping a base's unique ID to the site where it was allocated is created. Each time a new function is visited, the hashtable is cleared to prevent previously seen allocations in different functions from being erroneously reported when they're `free`d elsewhere.
The unique base ID is provided by *Frama-C*, and doesn't change throughout

the analysis, making it ideal to look up bases when they are `free`d to determine if they're short-lived.

Only allocations of a configurable maximum size or less are added to the hashtable to ensure that the only ones reported are those that can feasibly be stack allocated instead. Their location (filename and line number) is stored along with the statement they originated in so that the notification to the user can clearly indicate where changes are to be made.
For example, in the code listing in Section 2.2.1, the following is reported by *Forgetful* if it can determine that `alloc_size` is small enough to report.

```
[forgetful] Candidate for replacement in func: `free(
    alloced);` (pattern.c:4) frees base allocated at
`void *alloced = malloc((unsigned int)alloc_size);` (
    pattern.c:2)
```

On a `free`, *EVA* is used to determine what bases it could be freeing, and from there its ID is used to determine if the base identifies a small allocation and where it was allocated using the aforementioned hashtable.

Of course, not all allocations have a size that can be statically determined, with many instead having an interval as described in Section 3.3.2. In order to simplify application of the patch in cases where the allocation size is an interval, these are only reported if the maximum value of the interval is less than or equal to the configured maximum size to report. This decision results in the plug-in being unable to detect the case described by Stenberg in *cURL* [Ste17], as the allocation was not always below the size chosen for stack allocation.
Different behaviour for intervals could be added, to allow for cases where stack allocation or heap allocation are decided between at run-time. This would allow the plug-in to detect the case mentioned above.

It's also worth noting that bases are independent of the variable names they're allocated to. Take the following section, where `TOO_LARGE` and `SMALL_ENOUGH` are appropriately defined so as to ensure Forgetful doesn't and does report the allocations respectively.

```
1 int* val = malloc(TOO_LARGE);
2 if (rand() % 2)
3   val = malloc(SMALL_ENOUGH);
4 free(val);
```

Forgetful will report the allocation on line 3 even though `val` can also be too large to report. This is because `val`'s value is found by *EVA* to be a set of bases (from lines 1 and 3) where one is too large to report and the other is small enough, rather than describing it as a single base with a range which extends enough as to be too large.

This differs from the following example, in which there is only one base which has an interval too large to report as its size.

```
1 size_t size = TOO_LARGE;
2 if (rand() % 2)
3   size = SMALL_ENOUGH;
4 val = malloc(size);
5 free(val);
```

3.4.3 Difficulties Encountered

A number of difficulties were encountered throughout the project, described below.

Build and Installation

An initial and unexpected difficulty was the installation of *Frama-C* itself. A bug in one of its dependencies resulted in it being unable to compile, which made it impossible to install from source with no binaries available. A few patches (which all had to be applied together) for the issue were found online, not yet applied to the dependency despite having been available and known to the maintainer for a few months. To enable *Frama-C* to be built, the patches were applied during the build process.

This is more complicated than it sounds, as the build process downloaded new sources which needed to be patched as well, requiring a total of 3 or 4 patches which must be applied only once to any given file and before the next part of the build process started using those files. A script was eventually written to patch files on the fly once they were created, and run alongside the build process.

OCaml and *Frama-C* Learning Curve

As expected, there was a significant learning curve involved in learning both a new language, OCaml, and a new platform, *Frama-C*. As OCaml's has similarities to languages such as Java and Haskell, some of this learning curve was mitigated due to familiarity with those languages. Having no prior experience developing part of or even using a static analysis tool, *Frama-C* was still difficult but manageable to broach due to both the user manual [Fra08g] and the plug-in development guide [Fra08d].

Kernel and Core Plug-in Documentation

Despite *Frama-C*'s lengthy guides for both users and plug-in developers, its documentation suffers from a lack of structure and detail. The plug-in development guide walks the developer through the development of a basic hello world style plug-in, building up to the currently recommended plug-in architecture. Once the basic structure is attained, the guide branches into a handful of sample plug-ins, jumping between *Frama-C*'s capabilities in a somewhat incoherent manner. By this point enough information has been provided to start on development however, so the API documentation provided separately is generally sufficient.

One case in which the API documentation proved insufficient was in its description of bases. In particular, functions are provided that allow translation from a base to variable info, which initially seemed to provide a simple way to perform most of the analysis. A base could simply be retrieved at the `free` site, and have variable info retrieved from it to determine its allocation site and size. However, while variable info should include its allocation site, the variable

info produced from a base doesn't include valid location data. This is why allocations have to be tracked forwards from the `malloc` instead.

EVA's documentation was the most lacking, with the majority of its manual being dedicated to users interacting with *EVA* through ACSL or the *Frama-C* GUI. In the entire manual, there is only one mention of an *EVA* API, which is the `value` function. However, the `value` function only returns the value before a statement has been executed and the Forgetful plug-in needs access to the result after a `malloc` call. A solution was eventually found online [ano16], but searching for one led to significant delays.

3.5 Final State of the Plug-in

The state of the Forgetful plugin is most easily communicated by examining results from running the analysis on various code samples.

```c
#include <stdlib.h>

void* recurse_and_free(int recurse) {
  if (recurse) {
    free(recurse_and_free(recurse - 1));
    return malloc(1);
  } else {
    return malloc(1);
  }
}

void main() {
  free(recurse_and_free(2));
}
```

The above code listing produces no output from *Forgetful*, and the output from *EVA* indicates that the issue is with the function `recurse_and_free`. In particular, *EVA* can't determine whether the function terminates as it doesn't currently support recursion. Since *Forgetful* depends on *EVA*, it also can't analyse recursive functions.

The following code listing describes inline which allocations are reported as replaceable with stack allocation. The default max allocation size to report on was set to 48 bytes, based on the results discussed in Chapter 4.

```c
#include <stdlib.h>
#include <time.h>

int* delegate(size_t size) {
  // reported, as all functions are inspected
  int* distraction = malloc(sizeof(int));
  free(distraction);

  return malloc(size);
}

int main(void) {
  srand(time(NULL));
```

```
14
15  // this one is directly allocated and is reported
16  int* local = malloc(sizeof(int));
17  // this one's indirectly allocated, so it isn't reported
18  int* delegated = delegate(sizeof(int));
19
20  // this one is reported by setting max size to 64
21  int* capped = malloc(sizeof(int) * (rand() % 16 + 1));
22
23  // this one is reported by setting max size to 4 * 32768
24  // (RAND_MAX in frama-c + 1), i.e. 131072
25  int* randomly = malloc(sizeof(int) * (rand() + 1));
26
27  int* fixed = NULL;
28  if (rand() % 2) {
29    // this one is reported by default
30    fixed = malloc(sizeof(int));
31  } else {
32    // this one is reported if the max size is set to 4 * 10000, i.
      e. 40000
33    fixed = malloc(sizeof(int) * 10000);
34  }
35
36  // reported by default
37  int* varied = malloc(sizeof(int));
38  if (rand() % 2) {
39    // reported if max size is set to 4 * 17 i.e. 68
40    varied = malloc(sizeof(int) * 17);
41  }
42
43  free(local);
44  free(delegated);
45  free(randomly), free(capped);
46  free(fixed);
47  free(varied);
48  return 0;
49 }
```

Chapter 4

Case Studies

This chapter will include, for each case study: reasoning under which that particular case study was chosen; predictions of the results; the code for the benchmark, or a reference to where the code can be found; results of the benchmark; and a discussion of the results compared to the conclusion.

For the code used in the benchmarks, refer to Appendix A.

4.1 Specialised Benchmark, Sorting

The first benchmark involves minimal allocations and a large amount of busy work.

The benchmark setup code creates input data of an array of structs containing a random value and a sequential ID (which isn't used). It then calls each of the four benchmarking functions described below a large number of times (10,000 to 20,000 times) in a row each, timing how long each execution takes in total.

The benchmarking functions each perform a single allocation before copying the input buffer into the newly allocated block. How this allocation is performed is the core difference. The four types of allocation used are described in Table 4.1.

Note that `alloca`[1] is used despite its unsuitability for safe and portable code. This is to simplify ensuring that there is always enough space in the `stack` benchmark function, in which errors are not fatal.
Another caveat to the use of `alloca` is that while it appears to be a standard library function, it's not actually part of the ANSI/ISO-C standard [ISO11] and is simply replaced inline by compiler implementations. This causes portability concerns.

[1] `alloca` performs a stack allocation at runtime in a similar manner to `malloc`, but does not perform checks for whether the stack will overflow as a result and doesn't require `free`ing as restoring the stack pointer will automatically free the block

Allocation Type	Description
`malloc`	Performs the allocation with `malloc`, providing a base performance to compare against. `free`s the allocated block before returning
`stack`	Performs the allocation with `alloca` regardless of the size of the input data even though this would introduce a significant risk of stack overflow in real code. No `free` required
`dynamic`	Performs the allocation with `malloc` if the input data is larger than a given threshold (in this case, 64 items), otherwise uses an array of static size declared as a local variable (and, hence, stack allocated as part of the calling code). `free`s before returning if required
`external`	Performs no allocation, instead taking a parameter of a sufficiently large buffer to copy data to, which it doesn't `free`. Acts as a control as no allocation is performed and so it should be the fastest

Table 4.1: Allocation Types

4.1.1 Program Reasoning

This benchmark was chosen to provide a sort of floor for performance changes due to application of the patch, as a minimal amount of time should be spent in allocations to be gained back by more efficient allocations.

4.1.2 Predictions

It is expected that the `external` allocation method should be fastest, as it spends no time performing allocations.
The `stack` allocation method should be the next fastest, not requiring a `free` or finding of a suitable block by `malloc`, and should maintain a small performance increase even for larger input sizes, although this will be dwarfed by the time spent sorting.
`dynamic` should be roughly tied for performance with `stack` for small input sizes, thanks to not having the overhead of `alloca`, but adding the overhead of determining whether to use `malloc` or not. It also suffers the overhead of a `free` on exit, and should perform worse than the `malloc` allocation method on larger input sizes, but again this should be dwarfed by the time spent sorting.

4.1.3 Patch Code

Refer to the benchmark code in the `src/naive` directory in Appendix A.

4.1.4 Results

In the diagrams in Figure 4.1 through Figure 4.5, the y axis represents the time taken for a given allocation method compared to the `malloc` benchmark, while the x axis represents the size of the input data. Values below 1 on the y axis outperform `malloc`.

The values in the diagrams are taken from a number of runs, from which the top third and bottom third of values were dropped as outliers. The minimum, maximum and mean values were then taken from the remaining values.

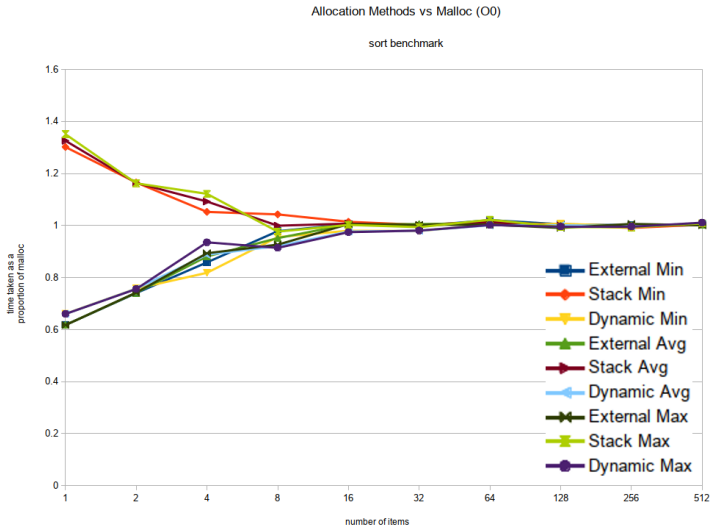

Figure 4.1: Unexpectedly, `stack` performs worse than `malloc`. Others perform as expected. From inspecting the generated assembly, it looks like -O0 produces very inefficient code for alloca.

4.1.5 Comparison to Predictions

Results largely matched the predictions, with the exception of `dynamic` outperforming `stack` narrowly. This may be due to the generated code for `alloca` being slower than the simple check required by the `dynamic` case.

These results bode well, especially since the lack of significant performance difference between `dynamic` and `stack` means that the safer option, `dynamic` can be used without issue in real code.

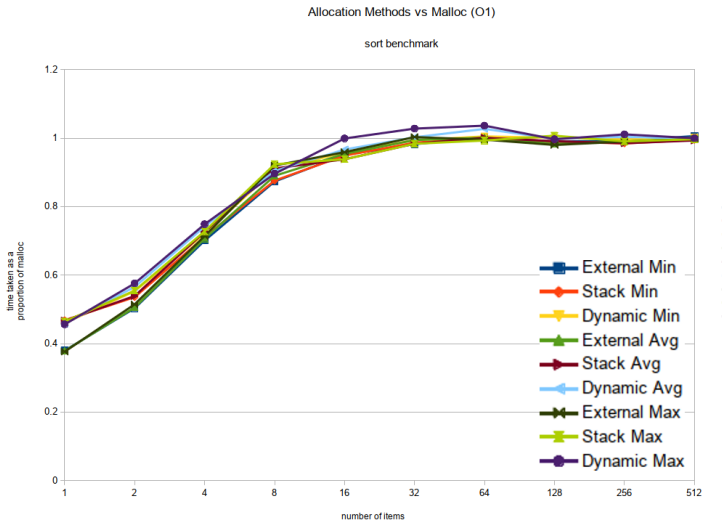

Figure 4.2: Everything performing mostly as expected. `stack` and `dynamic` seem to perform very similarly, with `dynamic` unexpectedly underperforming `malloc` before falling back to `malloc` level performance at around the same input size as the other methods' performance converge with malloc.

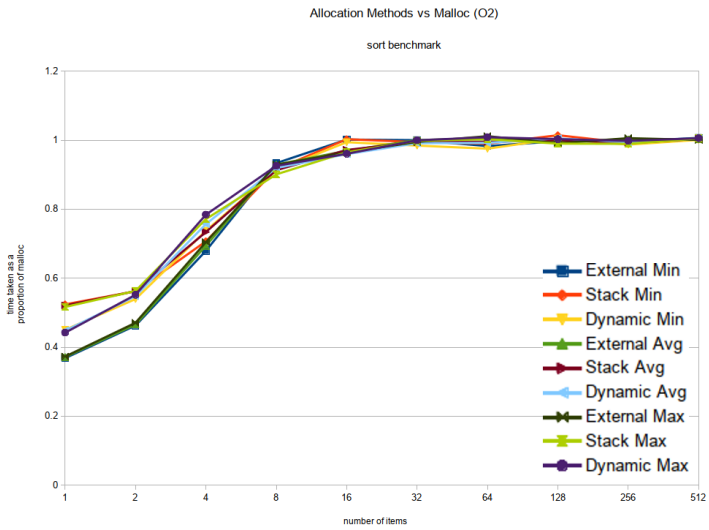

Figure 4.3: Everything mostly as expected. `dynamic` shouldn't be faster than `stack`, but the difference is small and the gap is closed after the first item in any case.

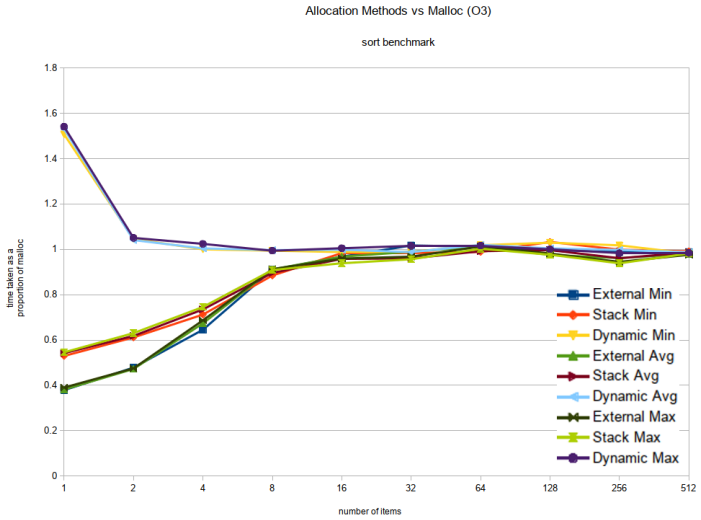

Figure 4.4: An unusual drop of performance in the `dynamic` case, with the rest performing as expected. This appears to be caused by function inlining, where the function copying input to the allocated buffer is inlined twice in the `dynamic` benchmark function. The copy used for the case using `malloc` is optimised to use `memcpy` while the other does not. This causes the `stack` case to perform worse due to the slow copy, rather than any allocation issue.

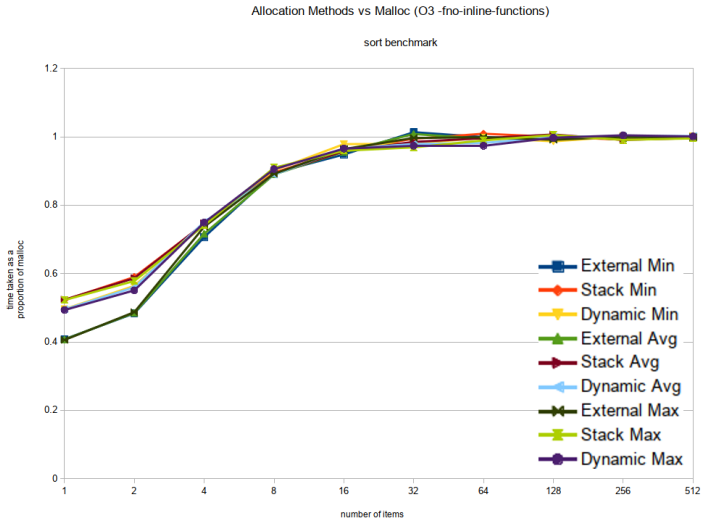

Figure 4.5: The same as the figure above, but with function inlining disabled. Once again, `dynamic` slightly outperforms `stack` on small cases.

4.2 Specialised Benchmark, Parallel Allocations

The second benchmark involves a large number of allocations and minimal busy work.

The setup code is identical to the setup for the sort benchmark described above, but instead of calling each function a fixed number of times it sets up a number of threads (in this case 4) which all call the function simultaneously for a total of 2 seconds. This is repeated a number of times (in this case 10) and the number of times the benchmark function was successfully called is taken as an indication of performance.

However, instead of sorting the input data, the functions simply copy the input data to the allocated buffer. The allocation types used are the same as in the sort benchmark.

4.2.1 Program Reasoning

This benchmark seeks to produce a ceiling for performance increases. While there are many implementations of `malloc`, most involve locks at some level of granularity. In particular, this benchmark was run using glibc's `malloc`, which does take locks internally [sou18]. This benchmark seeks to maximise the contention of these locks, allowing the benchmarking functions to avoid losing time to unavailable mutexes by avoiding `malloc`s, making them faster by comparison.

4.2.2 Predictions

The predictions are the same as in the sort benchmark, but the speed increases should be more significant.

4.2.3 Patch Code

Refer to the benchmark code in the `src/parallel` directory in Appendix A.

4.2.4 Results

In the diagrams in Figure 4.6 through Figure 4.8, the y axis represents the number of function calls completed for a given allocation method compared to the `malloc` benchmark, while the x axis represents the size of the input data. Values above 1 on the y axis outperform `malloc`.

The values in the diagrams are taken from a number of runs, from which the top third and bottom third of values were dropped as outliers. The minimum, maximum and mean values were then taken from the remaining values.

4.2.5 Comparison to Predictions

Similarly to the sort benchmark, results mostly matched the predictions. As expected, performance increases were much more significant than in the sort

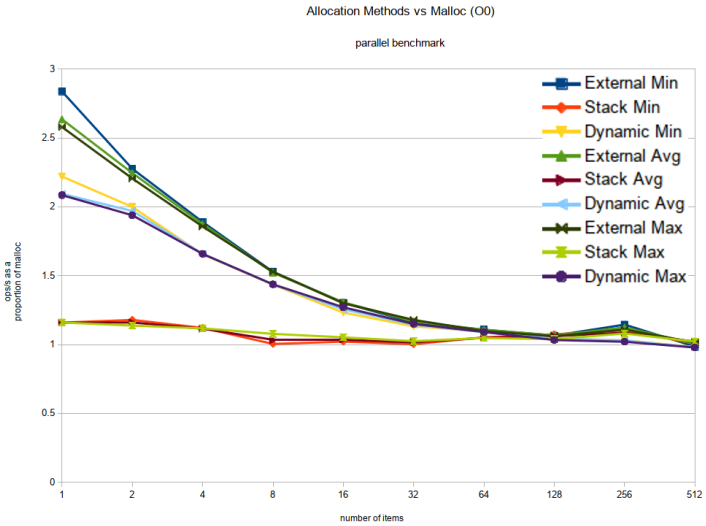

Figure 4.6: `stack` performs poorly again, although this time it still outperforms `malloc`. Other allocation methods perform as predicted.

benchmark. This is expected, as the changes made here are essentially the same as replacing a locking algorithm with a wait-free algorithm.

However, the situations in which this sort of performance increase would be possible are quite limited, as programs are not often allocating at such a high rate.

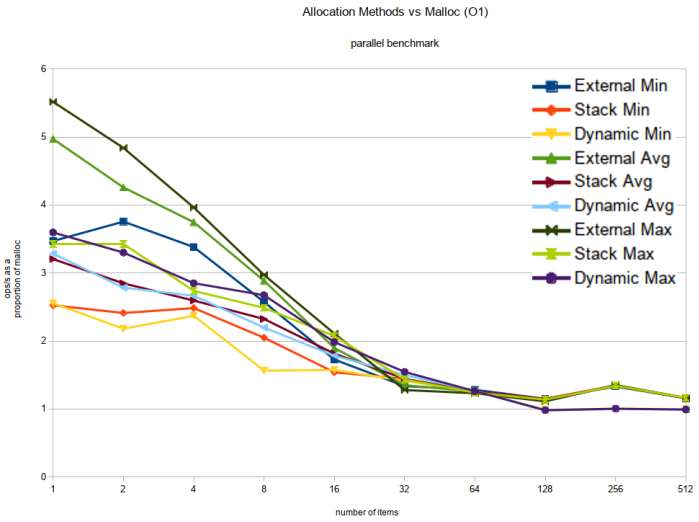

Figure 4.7: As seen in the sort benchmark, `stack` and `dynamic` perform roughly the same, while external outperforms both. Unlike the sort benchmark, `stack` and `external`'s performance benefits don't completely drop off on large input data.

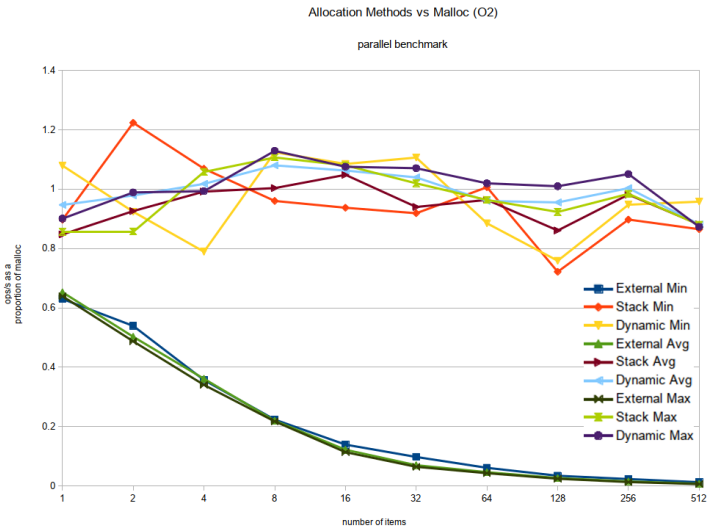

Figure 4.8: At this optimisation level the compiler was able to determine that the results of the copying in the benchmark functions were not being used, so the entire functions got optimised out other than the `external` method, which essentially degenerates to a `memcpy`. Higher optimisation levels yield the same result.

4.3 Real World Benchmark, *cURL* Patch

The third and last benchmark involves a more careful analysis of Stenberg's patch to *cURL* [Ste17]. It covers both parts of the change described in Section 2.1.

The benchmark is performed by cloning *cURL*'s source and checking out[2] the commits in which the changes were made.

In order to test the linked list changes, simply running regular *cURL* requests is enough, as indicated by the blog post. For the polling function changes, one of the code samples distributed with the *cURL* source code exercises the exact function to be benchmarked, `curl_multi_wait`.

4.3.1 Program Reasoning

Clearer figures for the performance impact on a real-world program were needed. The two previous benchmarks demonstrate theoretical improvements in carefully constructed scenarios, but to gain a better impression of the real-world impact, real-world code must be tested.

Since the blog post lumps in all changes from some 230 or so commits, it can't be assumed to only be a result of the allocation changes. The goal of this benchmark is to isolate the changes with their effects.

4.3.2 Predictions

Given the results of the previous benchmarks and from the blog post, results are expected to be noticeable in the direction of outperforming `malloc`, but smaller than described in the blog post.
This is accounting for the fact that allocations alone were unlikely to account for 30% of the execution time as suggested in the blog post. While a large speed-up was shown in the benchmarks, they were designed primarily around allocations and took care to only measure times at the sites where allocations were changed.

By contrast, *cURL*'s timing cannot focus in only on allocation sites, and if it did focus on only the allocation sites it could miss knock-on effects from the change.

Note that the polling function, `curl_multi_wait`, is benchmarked with 1 and 11 file descriptors. The single file descriptor benchmark is intended to trigger the best case, where a small heap allocation is replaced with a stack allocation, whereas the 11 descriptor benchmark is intended as a control, as its performance should be identical before and after the change. This is due to the fact that only up to 10 descriptors are stack allocated.

There are a number of changes included in the linked list changes, such as: changing initialisation functions from a `malloc`+`memset` to a single `calloc` having the same effect; or a cascading effect from the change to the linked list

[2] A checkout in this context refers to `git-checkout`, whereby the state of the code is rewound to a given point

Time Type	Description
real	Wall clock time elapsed, i.e. actual time taken from the program starting to when it exits
user	CPU time spent outside of kernel code
sys	CPU time spent in kernel code

Table 4.2: Description of the meanings of times reported by *time*

functions being unable to fail due to a lack of memory resulting in less checks being necessary in code using those functions. These changes could all impact the performance, but should favour the new version of the code performance-wise.
To test the linked list changes, a single large and empty file was retrieved from a server running locally.

4.3.3 Patch Code

Refer to the `replicate.sh` script in the `src/curl` directory in Appendix A. Running this script on a machine with the required dependencies (refer to the *cURL* source to determine what dependencies are required to build it) will produce the executables used to benchmark and run the benchmark.

4.3.4 Results

In the diagrams in Figure 4.9 through Figure 4.11, the y axis represents the run-time of the patched version of *cURL* as a proportion of the time taken to perform the same task by the parent commit. This is used as a proxy for performance. On the x axis, `curl-llist` is the result of testing the linked list changes, `curl-multi-wait-1` is the result of testing the polling function changes with a single file descriptor, `curl-multi-wait-11` is the control using 11 file descriptors.

The min/avg/max times are as reported by the *time* utility, and a description of the meaning of user/real/sys is in Table 4.2.

The values in the diagrams are taken from a number of runs, from which the top third and bottom third of values were dropped as outliers. The user, real, and sys time values were then taken from the remaining values and their minimum, maximum and mean found.

Values less than 1 outperform the parent commit.

4.3.5 Comparison to Predictions

The results in this benchmark were unexpectedly poor, and differed completely from predictions and indeed from what Stenberg claimed. It seems that one or more of the other commits lumped in with his measurements may have affected the performance difference.

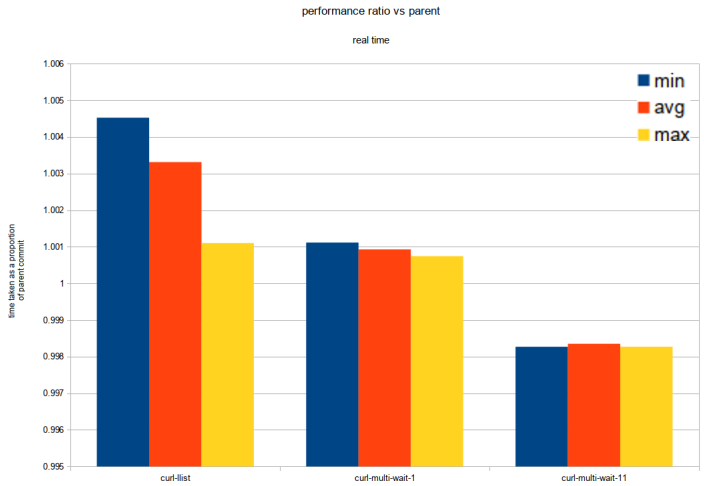

Figure 4.9: Nothing varies by more than 0.5%. The only benchmark that performs better is the one that should perform identically, suggesting that any performance benefit is shadowed by other effects.

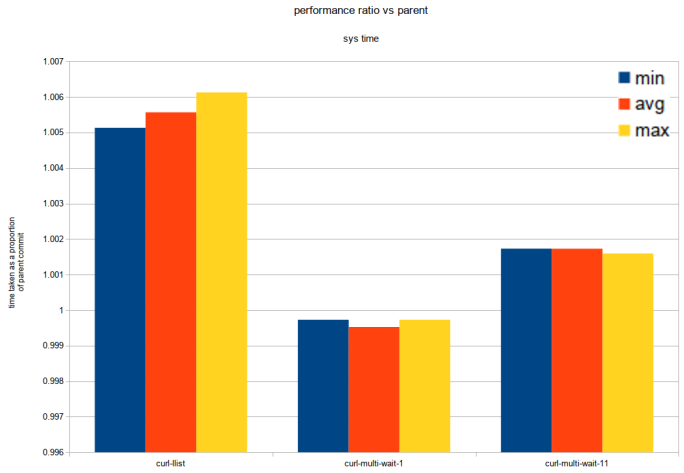

Figure 4.10: sys time should in theory decrease in the first two cases, as less `mallocs` should mean less time spent in kernel code (such as `brk`/`sbrk`/`mmap`). Once again, the maximum variation is 0.6%, but in the wrong direction compared to predictions.

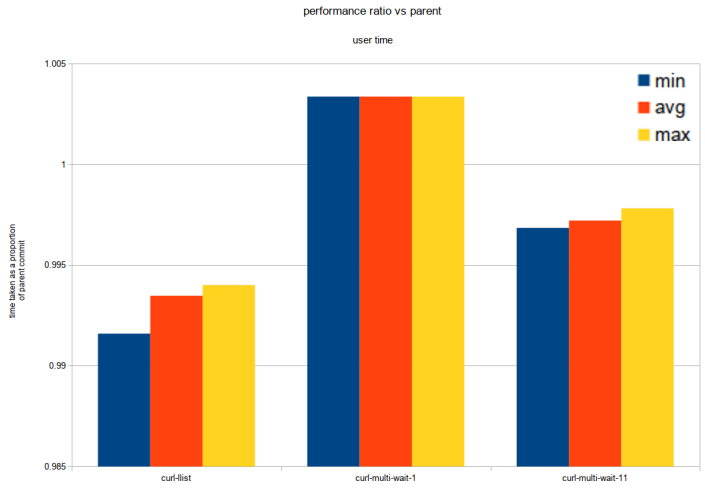

Figure 4.11: The llist case should decrease in time thanks to simplified code due to removal of `mallocs` and use of more efficient library functions (`calloc` instead of `malloc` and `memset`). For the multi-wait case, user time shouldn't really change. This is mostly met, particularly in the llist case.

Chapter 5

Conclusion

This chapter will discuss the results in the proper context to determine whether they show merit as a worthwhile optimisation, and also discuss the state of the plug-in in further detail, in particular discussing its limitations.

5.1 Results

There are two main categories of results to be discussed, the specialised benchmarks and the real world study.

5.1.1 Specialised Benchmarks

The specialised benchmarks, while constructed to be an ideal case for application of the patch, clearly validate that the patch can result in a significant performance increase in specific cases.

That being said, they also serve as a reminder that even in simple code there may be hidden issues and complexity, such as `alloca`'s performance at the `O0` optimisation level in both of the benchmarks, or the unexpected slowdown of the `dynamic` allocation method at the `O3` optimisation level in the sort benchmark.

The benchmarks also highlight that, in certain cases, the patch makes no performance difference (little to no difference by 16 items in the sort benchmark, and minimal at the changeover point of 64 items in the parallel benchmark).

Both the `dynamic` and `stack` allocation methods as implemented have downsides that are present even when their benefits are no longer being reaped. Note that the following all assume correct usage, for example ensuring that no pointer to the allocated buffer escapes the function scope, or is made accessible to another thread (as threads do not share stacks [Krz14]).

`dynamic` Method Downsides

The stack frame is bloated with the buffer even when it's not being used, as space must always be allocated for it even if it's determined at run-time that

the buffer can't be used.

Depending on where the buffer is declared, whether multiple buffers are declared (if there are multiple inputs that can be copied to stack allocated buffers), and how large the buffers are, locality of the stack could potentially be affected.

The larger stack frames required also increase the likelihood of stack overflow occurring, which will generally result in a segmentation fault due to the invalid access to storage (outside the bounds of the stack) [ISO11].

`stack` Method Downsides

The most immediate issue with the `stack` method as implemented is that it unconditionally uses `alloca` to allocate the buffer. As mentioned before, `alloca` is not in the C standard and as such is both machine and compiler dependent.

The unconditional allocation also means that the risk of stack overflow, and hence a segmentation fault, is even greater than in the `dynamic` case, due to use of `alloca` even if the allocation to be performed is very large.

Another issue with `alloca` is that it has no way to indicate if the stack frame cannot be extended. Unlike `malloc`, it never returns `NULL`. This means that is no way to tell if an allocation was successful until the buffer is used, causing a stack overflow and segmentation fault.

Handling The Downsides

A better solution than the `dynamic` or `stack` allocation methods may be somewhere in between the two. Assuming that the lack of portability of `alloca` is not an issue, some of its benefits can be used to eliminate some downsides of the `dynamic` method. This is referred to as the `hybrid` allocation method hereafter.

Rather than performing a check on the input size and using a statically declared buffer on the stack, the stack allocation itself can be performed with `alloca` (or, under the C99 standard, a variable length array could be used with portability benefits). This would prevent bloating the stack unnecessarily, allowing only the buffers that will actually get used to be stack allocated.

To handle the increased risk of a segmentation fault, a signal handler could be written capturing the `SIGSEGV` signal raised. However, signal handlers are extremely restricted in what they can do, which functions they can call, and even which variables they can write to [Car17]. The increase is complexity resulting from needing to write and maintain this signal handler, if a signal handler can even be written to sufficiently generically resolve stack overflow errors[1], is unlikely to be justified by any performance benefits gained.

[1] A trivial, but not necessarily useful, handler simply expands the stack when a stack overflow is detected, but this is unlikely to be possible to do for any extended amount of time, and only works in an environment can be extended at runtime using only functions permissible in signal handlers. This also requires the handler to be able to distinguish between a signal raised due to a stack overflow or due to another reason, such as a `NULL` dereference.

5.1.2 Real World Benchmark

The real world benchmark had disappointing results when compared to the specialised benchmarks, and even to Stenberg's original claims. Results were found to be indistinguishable from variance in the test environment itself, in sharp contrast with the 30% performance increase claimed [Ste17].

Despite these results, this doesn't mean that the patch is not worth applying, or at least researching further. Compiler optimisations are known to produce minimal performance benefits individually, instead building up mutually to more significant gains over time and in conjunction with each other.
This observation was noted in Proebsting's Law, in which Proebsting claimed the following, with the figure of doubling every 18 years chosen to match the more well-known Moore's Law.

> These assumptions lead to the conclusion that compiler optimization advances double computing power every 18 years. [...] compiler optimizations contribute only 4% [per year]. Basically, compiler optimization work makes only marginal contributions. [Pro98]

These findings were formalised 3 years later by Scott, in which the rate of improvement was estimated to be closer to 2.8–3.6% per year, or even as high as 4.9%, depending on what year was taken as the starting point in which compiler optimisations begun to be developed [Sco01]. Scott also notes that there's no reason to give up on compiler optimisations, as any performance benefit is better than none, and in particular there's no reason to turn down existing performance benefits.

5.2 State of the Plug-in

The current state of the *Forgetful* plug-in is described in Section 3.5, where the instances of the pattern it can detect are shown and some of its limitations discussed. Some of these limitations are outside of the control of the plug-in, such as the inability to process functions which use recursion, which is imposed by *EVA*. Other limitations, such as only performing intra-procedural analysis, or the simplified handling of intervals, are internal limitations.

5.2.1 Interval Handling

As mentioned in Section 3.4.2, the handling of allocations whose size is some value from an interval is simplistic, with the allocation only being tracked if the entire interval is smaller than the configured maximum allocation size.

As an alternative, intervals could be accepted if some configurable portion of the interval was smaller than the maximum allocation size. This information could then be used in conjunction with profiling data to determine whether the introduction of the `hybrid` allocation method or similar would be worthwhile.

5.2.2 Intra-Procedural Analysis

Forgetful is currently limited to intra-procedural analysis, due to the additional complexity that inter-procedural analysis would add and the limited time available to complete the project. Tracking the progress of allocated blocks of memory across function boundaries could significantly improve the quality and number of sites that *Forgetful* can report.

Consider the following code listing:

```
1  void* xmalloc(size_t size) {
2    void* block = malloc(size);
3    if (!block) exit(1);
4    return block;
5  }
6
7  void main(void) {
8    void* unreported = xmalloc(1);
9    free(unreported);
10 }
```

Currently, *Forgetful* cannot detect that `unreported` is a candidate for replacement, despite `xmalloc` being a simple wrapper on `malloc` which cannot return `NULL`. By tracking allocations across function calls, this case could be detected.

This pattern is not unusual, and is similar in structure to a constructor function which allocates and initialises a struct, again being quite a thin wrapper on `malloc` with some setup steps on the returned value.

Barrett and Zorn defined a term for function invocations which eventually led to a `malloc`, calling them length-N call chains, where a length-1 call chain is a direct invocation of `malloc`. While researching the ability to predict lifetimes at allocation sites, they discovered that length-4 call chains were almost always sufficiently long to determine the lifetime of a given allocation [BZ93].
Their methods and goals were different to this project's, and their figures were found experimentally with a limited number of programs 25 years ago, but their findings suggest that analysing longer call-chains would be of worth.

5.2.3 Notification Output

The output displayed in Section 3.4.2 is real, but is not representative of *Forgetful*'s output. As analysis is performed on the normalised AST, it may occasionally reference statements that do not exist in the source code itself. Consider the following code listing:

```
1  void main(void) {
2    void* alloced = malloc(1) + 1;
3    free(alloced - 1);
4  }
```

The statement on line 2 is rewritten to the following:

```
1  void main(void)
2  {
```

```
3   int tmp;
4   tmp = malloc(1);
5   void *alloced = (void *)(tmp + 1);
6   free(alloced - 1);
7   return;
8 }
```

This means that when the allocation is reported, it reports the contents of line 4 as the allocation site, which makes it difficult for the user to discern what allocation is being referred to, as can be seen in the following output from the plug-in.

```
[forgetful] Candidate for replacement in main: `free(
    alloced - 1);` (samples/transformedOut.c:5) frees base
     allocated at `
     tmp = malloc((unsigned int)1);` (samples/
         transformedOut.c:4)
```

The formatting of this output raises another issue that could be revisited, which is that the output is not easily machine parsable. This could make it less useful in automated pipelines using the tool to detect issues in pull requests, for example.

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, and the optimisation field is endlessly expandable due to the *full employment for compiler writers* theorem [App04]. To that end, listed below are some ideas for potential improvements on the *Forgetful* plug-in and in the general space of compiler optimisations.

7.1 Further Work on Allocation Changes

This section discusses potential further work directly on the *Forgetful* plug-in, and also in the general area of memory allocation research.

7.1.1 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, but this would require the existence of 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 Infer [Fac13] static analyser, so future work could also involve extending that platform instead.

7.1.2 Automatically Performing Fixes

Ideally, fixes would be automatically generated and patched into the code as a pre-compilation step, avoiding added complexity from the programmer's point of view while still taking advantage of performance and memory benefits.

Potential intermediate steps toward that goal could involve generation of patches that could be manually applied to code before compilation, introducing the optimisation.

Based on the documentation for *Frama-C* and one of its plug-ins, *PathCrawler*, it is unclear whether there are code generation capabilities in *Frama-C* or if the plug-in introduced them itself. Depending on this, this goal may be very complicated to complete, and potentially not worthwhile.

7.1.3 Higher Precision in Allocation Sizing Calculations

Frama-C has built in defaults for sizes of various types, which mostly seem to be based on a 32-bit architecture. As a result, size calculations can be inaccurate, particularly on 64-bit machines with factor of 2 inaccuracies in any size calculation involving pointers.

Not accounting for these differences will lead to *Forgetful* recommending replacements at sites where more memory than the configured maximum is allocated. *Frama-C* provides a system to define sizes of all the types, which could be exposed to users to allow them to specify their architecture details.

7.1.4 Alternative 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.

7.2 General Compiler Optimisation Research

This section is concerned with the more general area of research into compiler optimisations, both in terms of optimisations to be researched or implemented, and also in terms of research into measurement of optimisations.

7.2.1 Potential For Bias Against Optimisations In Proebsting's Law

Scott notes four potential issues with his analysis of Proebsting's Law. While the first three are suitably discussed and dismissed, the fourth is briefly covered

and then assumed to not disproportionately affect the analysis. The issue is that it is impossible to fully disable all optimisations in a compiler through configuration.

The method used involves compiling certain benchmarks first with all compiler optimisations disabled through configuration options, then again with the flags recommended by the vendors for those benchmarks. The assumptions are that a compiler with all optimisations disabled is equivalent to a compiler before new optimisations were discovered and implemented. However, certain optimisations such as peephole optimisations or more efficient code generation techniques cannot be disabled. This results in a baseline program which is more efficient than it would have been, had it been compiled with an older compiler [Sco01].

As a result of a more efficient baseline program, the relative improvement is lower, biasing the results against higher performance increases.

Further work determining a more accurate and general figure (or figures) for year-on-year improvements that can be reasonably expected of compiler optimisations would provide a point of reference to determine whether a given optimisation is worth introducing to industrial compilers.

7.2.2 Disproportionate Improvement of Optimisations

Proebsting's Law seeks to make a general prediction about performance improvements due to compiler optimisations over time but, as Scott notes, different types of benchmarks display very different levels of performance increase.

It is noted that floating point benchmarks improve significantly more than integer based benchmarks, which means that scientific applications will likely disproportionately benefit scientific applications [Sco01].

Similar to the previous section, perhaps determining expected figures for various areas of research could be worthwhile.

7.2.3 Targets for Optimisation

Contrary to the usual definition, *target* here does not refer to a target machine or architecture, but instead the goal of a given optimisation and the research into it.

In declaring his law, Proebsting also suggests that future research be directed at improving programmer productivity, rather than micro-optimisations [Pro98]. Pugh suggests the same, stating that optimisation of higher level constructs and designs have more space for optimisation than expression and statement-level optimisations.

Alongside these suggestions, Pugh lists a number of areas in which research may be worthwhile, including the following [Pug01]:

- Making higher level constructs and data types as efficient as primitives, with little to no programmer intervention

- Optimising safety checks away, so that safe code can be as performant as unsafe alternatives
- Performing optimisations that allow clean and easily readable code to be as efficient as carefully hand optimised code using techniques such as loop unrolling and cache-aligning related data (this is already performed to certain degrees in industrial compilers, but can require programmer intervention and cluttering code to maximise performance [GCC14])
- Allowing language level guarantees about certain optimisations and performance, such as guaranteeing that tail-call elimination will occur (this specific case is tentatively offered by Haskell [Wik15])
- Replacing inefficient algorithms with efficient versions having the same results, such as detecting presence of a quadratic stable sort and replacing it with a linearithmic stable sort
- Providing stronger safety guarantees for concurrent programs

7.2.4 Benchmark Design

Benchmarks are often designed, as the specialised ones in this project were, to elicit certain behaviours. As such, they are carefully tailored, hand optimised, and written at a very low level to avoid other behaviours that may affect their performance. However, this means that these benchmarks do not reflect real world code, and Pugh even argues that their style is so poor and unrealistic that they shouldn't be exposed to undergraduates [Pug01].

Instead, he suggests, they should be written to reflect that real code is complex, and should use techniques and utilities such as higher level constructs, parallelism and more modular components. This would enable benchmarking of bigger picture optimisations that the preceding sections all eventually build towards.

Bibliography

[BZ93] David A Barrett and Benjamin G Zorn. "Using lifetime predictors to improve memory allocation performance". In: *ACM SIGPLAN Notices*. Vol. 28. 6. ACM. 1993, pp. 187–196.

[RHR94] Design Patterns RHRJJV Erich Gamma. *Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

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

[Pro98] Todd A. Proebsting. 1998. URL: http://proebsting.cs.arizona.edu/law.html (visited on 04/26/2018).

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

[Pug01] William Pugh. *Is Code Optimization Relevant?* 2001. URL: www.cs.umd.edu/~pugh/IsCodeOptimizationRelevant.pdf (visited on 04/27/2018).

[Sco01] Kevin Scott. *On Proebsting's law*. Tech. rep. Technical Report CS-2001-12, Department of Computer Science, University of Virginia, 2001.

[App04] Andrew W Appel. *Modern compiler implementation in C*. Cambridge university press, 2004.

[LLV07] LLVM. *clang-analyzer Static Analyzer*. 2007. URL: http://clang-analyzer.llvm.org/ (visited on 04/18/2018).

[Fra08a] Frama-C. *ANSI/ISO C Specification Language*. 2008. URL: http://frama-c.com/acsl.html (visited on 04/23/2018).

[Fra08b] Frama-C. *Evolved Value Analysis User Manual*. 2008. URL: http://frama-c.com/download/frama-c-value-analysis.pdf (visited on 04/20/2018).

[Fra08c] Frama-C. *Frama-C Architecture*. 2008. URL: http://frama-c.com/plugins.html (visited on 04/20/2018).

[Fra08d] Frama-C. *Frama-C Plug-in Development Guide*. 2008. URL: http://frama-c.com/download/frama-c-plugin-development-guide.pdf (visited on 04/20/2018).

[Fra08e] Frama-C. *Frama-C Silicon has been released!* 2008. URL: http://blog.frama-c.com/index.php?post/2016/12/13/Frama-C-Silicon-has-been-released (visited on 04/20/2018).

[Fra08f] Frama-C. *Frama-C Static Analyzer*. 2008. URL: http://frama-c.com/ (visited on 04/18/2018).

[Fra08g] Frama-C. *Frama-C User Manual*. 2008. URL: http://frama-c.com/download/user-manual-Sulfur-20171101.pdf (visited on 04/20/2018).

[ISO11] ISO. *ISO/IEC 9899:201x Standard Programming languages – C*. 2011. URL: http://port70.net/~nsz/c/c11/n1570.html (visited on 04/24/2018).

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

[GCC14] GCC. *Loop-Specific Pragmas*. 2014. URL: https://gcc.gnu.org/onlinedocs//gcc/Loop-Specific-Pragmas.html (visited on 04/27/2018).

[Krz14] Paul Krzyzanowski. *Threads, Multiple flows of execution within a process*. 2014. URL: https://www.cs.rutgers.edu/~pxk/416/notes/05-threads.html (visited on 04/25/2018).

[Wik15] Haskell Wiki. *Tail Recursion*. 2015. URL: https://wiki.haskell.org/Tail_recursion (visited on 04/27/2018).

[ano16] StackOverflow User anol. *Getting result of value-analysis*. 2016. URL: https://stackoverflow.com/a/36138145/6519610 (visited on 04/23/2018).

[Car17] CERT Division Carnegie Mellon University Software Engineering Institute. *SEI CERT C Coding Standard Rule 11. Signals*. 2017. URL: https://www.cs.rutgers.edu/~pxk/416/notes/05-threads.html (visited on 04/25/2018).

[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).

[Vil17] Jules Villard. *Finding Inter-Procedural Bugs at Scale*. 2017. URL: https://mirrors.dotsrc.org/fosdem/2018/UD2.119/code_finding_inter_procedural_bugs_at_scale_with_infer_static_analyzer.mp4 (visited on 04/18/2018).

[sou18] glibc source. *glibc malloc source*. 2018. URL: https://code.woboq.org/userspace/glibc/malloc/malloc.c.html (visited on 04/24/2018).

Appendix A

Source Code

Refer to the attached CD, or to the project repository on GitHub (https://github.com/dariota/forgetful/) to obtain the source code.