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

1 Introduction **1**
- 1.1 Report Structure . . . 1

2 Background and Objectives **3**
- 2.1 Background on the Patch . . . 3
  - 2.1.1 Linked List Changes . . . 3
  - 2.1.2 Polling Function Changes . . . 4
  - 2.1.3 Results of the Changes . . . 4
- 2.2 Objectives of this Project . . . 5
  - 2.2.1 Tool to Detect Potential Patch Sites . . . 5
  - 2.2.2 Determining the Patch's Performance Impact . . . 6

3 Implementation **7**
- 3.1 Goals of the Plug-in . . . 7
  - 3.1.1 Non-Functional Goals . . . 7
  - 3.1.2 Functional Goals . . . 8
- 3.2 Static Analysis Platform . . . 8
- 3.3 The *Frama-C* Platform . . . 8
  - 3.3.1 Source Code Processing . . . 9
  - 3.3.2 The *Evolved Value Analysis* Plug-in . . . 10
- 3.4 Development of the *Forgetful* Plug-in . . . 12
  - 3.4.1 Visitor Pattern . . . 12
  - 3.4.2 Allocation Tracking . . . 13
  - 3.4.3 Difficulties Encountered . . . 14
- 3.5 Final State of the Plug-in . . . 14
  - 3.5.1 Identified Sites . . . 14
  - 3.5.2 Limitations . . . 14

4 Case Studies **15**
- 4.1 Case Study 1 . . . 15
  - 4.1.1 Program Reasoning . . . 15
  - 4.1.2 Predictions . . . 15
  - 4.1.3 Patch Code . . . 15
  - 4.1.4 Results . . . 15

4.1.5 Comparison to Predictions 15
4.1.6 Hypothesis 15

5 Conclusion **16**
5.1 Results 16
5.2 State of the Plug-in 16
5.3 Benefit of Further Work 16

6 State of the Art **17**
6.1 State of the Art 17
6.2 Similarity of the Patch to Concepts in Generational Garbage Collection 17
6.3 Predictions on Results of Generalisation of the Patch 17

7 Future Work **18**
7.1 Detecting Arbitrary Memory Allocations 18
7.2 Automatically Performing Fixes 19
7.3 Studying Other Architectures 19

Bibliography **20**

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, 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

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.

However, this comparison attributes all performance and allocation differences to these two commits, despite there having been 231 commits in total

[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.

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

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. A decision was made, largely for reasons of pragmatism and convenience, to follow the second approach.

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 amount of false positives wherein the plug-in suggests a site where 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 even 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

Notifying the user is a fairly self-explanatory goal, as there's no point detecting an issue and not noting it. 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 reasonable decision about what action to take.

The diff patch is more complicated, but would be incredibly useful. If the plug-in's 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 activity of their development and popularity (as a proxy for likelihood to be well supported and modern). The short-list which the eventual target platform was chosen from consisted of *clang-analyzer* [LLV07], *Frama-C* [Fra08e], and *Infer* [Fac13].

clang-analyze is written in C++, matching the *clang* codebase in 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 tools that were not chosen are discussed in further depth in Chapter 6 in comparison to *Frama-C* in a retrospective manner.

3.3 The *Frama-C* Platform

The static analysis platform chosen was *Frama-C*. *Frama-C* has an emphasis on 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 thanks to the first functional goal, it assists in reducing false positives thanks to its conservative approach and care around sites of potential undefined behaviour [Fra08f]. Additionally, that specification language is used by the

[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

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 [Fra08b].

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)

```c
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*

```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));
  tmp_0 = fc(1);
  tmp_1 = (int **)malloc(sizeof(point) * (unsigned int)tmp_0);
  int **ppoint = tmp_1;
  __retres = 2 * i;
  return __retres;
}
```

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 itself as provided to plug-ins to traverse is also annotated. It can be annotated in the source code itself, using *Frama-C*'s ACSL to add specifications, or annotations can be added by other plug-ins as they discover properties of the code [Fra08c].

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, enums, and some other miscellaneous things.

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 [Fra08a].

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:

```
int main(void) {
  srand(time(NULL));
  int randVal = rand();
  int randPrime = 2;
  int morePrimes[NUM_PRIMES];
  int index;
  int current = 1;

  for (index = 0; index < NUM_PRIMES; current++) {
    if (isPrime(current)) {
      if (rand() % 2) {
        randPrime = current * 4;
      }
```

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}		

```
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 following values at the end of the loop

In particular, note that each item in the array `morePrimes` is tracked separately by *EVA*, that `randPrime` can take on any of the prime values, that `randVal` can take on any values between 0 and *Frama-C*'s `RAND_MAX`, and that 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. The values reported are now

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

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 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 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 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 again degenerate its set of values into [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 [Fra08d]. 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 [Fra08c]. The guide outlines some common use-cases, providing some code samples and best practices. Parts of these are used together to create 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*.

The visitor classes 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 [Fra08c]) 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 as such 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 occurs 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.

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. However, this means that the plugin cannot 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.

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 the plugin doesn't and does report the allocations respectively.

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

The plugin 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, where there is only one base which has a size which is an interval too large to report.

```c
size_t size = TOO_LARGE;
if (rand() % 2)
  size = SMALL_ENOUGH;
val = malloc(size);
free(val);
```

3.4.3 Difficulties Encountered

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

3.5 Final State of the Plug-in

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

3.5.1 Identified Sites

3.5.2 Limitations

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 Plug-in

What state is the plug-in 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 plug-in) 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* plug-in 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

[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).

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

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

[Fra08a] Frama-C. *Evolved Value Analysis User Manual*. 2008. URL: http://frama-c.com/download/frama-c-value-analysis.pdf (visited on 04/20/2018).

[Fra08b] Frama-C. *Frama-C Architecture*. 2008. URL: http://frama-c.com/plugins.html (visited on 04/20/2018).

[Fra08c] 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).

[Fra08d] 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).

[Fra08e] Frama-C. *Frama-C Static Analyzer*. 2008. URL: http://frama-c.com/ (visited on 04/18/2018).

[Fra08f] Frama-C. *Frama-C User Manual*. 2008. URL: http://frama-c.com/download/user-manual-Sulfur-20171101.pdf (visited on 04/20/2018).

[Fac13] Facebook. *Infer Static Analyzer*. 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).

[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).