Escape Analysis for Static Single Assignment Form

Luigi D. C. Soares[1], **Luís F. W. Góes**[1]

[1]Institute of Exact Sciences and Informatics (ICEI)
Pontifical Catholic University of Minas Gerais (PUC-MG)
Belo Horizonte, MG - Brazil

luigi.soares@sga.pucminas.br, lfwgoes@pucminas.br

Abstract. *In this work we proposed a type of static analysis called Escape Analysis, which aims at identifying cases where it is safe to allocate in the stack an element that would initially be stored in the heap. In the approach presented here, this analysis is being done by means of building and traversing a directed graph G. Each edge of the set $E \in G$ connects a value v of a function with the values that contains v as an argument (i.e. places where v is being used). We tested the algorithm with one synthetic and four real benchmarks. The results show that the overhead introduced by the garbage collector can cause significant impacts on the response times of some applications.*

Resumo. *Neste trabalho, foi proposto um tipo de análise estática conhecida como Escape Analysis. Esta possui como objetivo a identificação de situações nas quais é seguro alocar na pilha um determinado elemento que antes seria armazenado no heap. Na abordagem apresentada, a análise é realizada a partir da construção de um grafo direcionado G. Uma aresta do conjunto $E \in G$ conecta um valor v de uma determinada função com valores que possuem v como argumento (i.e. locais onde v é utilizado). Cinco benchmarks diferentes (um sintético e quatro reais) foram utilizados para avaliar o algoritmo proposto. Os resultados mostraram que a sobrecarga introduzida pelo coletor de lixo pode causar impactos significativos nos tempos de resposta de algumas aplicações.*

1. Introduction

Memory area reserved for program data is typically composed by three distinct regions of: (a) code; (b) global/static data; and (c) dynamic allocation. The last one is defined as dynamic because its size changes during runtime. It is divided into a stack, which works in a LIFO (last in, first out) manner, and a heap[1] [Louden 1997]. They differ by two main characteristics: performance and knowledge about lifetime of each object.

Access to data allocated in the stack area is extremely fast; in the heap, however, it is slower and therefore requires efficient management so this does not become a bottleneck. Every item pushed into the stack is popped when it leaves the scope. These instructions are determined at compile-time. On the other hand, a value stored in the heap has unknown lifetime. Thus, they remain alive until being explicitly destroyed.

[1]It should be noted that this is not related to the heap data structure. It is named this way for historical reasons.

Some language implementations imply that heap management must be done by the programmer (e.g. in C, with malloc and free). This manual task is complex and can cause multiple errors. Among them are: dangling pointers, which are objects that do not hold reference to a valid location; memory leaks, which refer to allocated areas without valid pointers, making them impossible to be released; and double free (calling free on a pointer twice).

For this reason, many languages implement automatic dynamic allocators. Their main objective is to collect objects no longer accessible from program variables. This has traditionally been done either by garbage collection or reference counting. Garbage collection involves a periodic disruption of program execution. Reference counting, whereas less disruptive, normally requires substantial storage overhead. Hence, a good heap manager should utilize memory space effectively and satisfy requests in as few instructions as possible [Barth 1977, Iyengar 1996].

Although less common than the cited techniques, there are other ways to deal with automatic memory management. Some of them are based on static analysis, thus eliminating runtime costs of space and time. The present work intends to reduce this overhead by transferring a slice of that to compile-time through analysing intermediate code produced by the compiler.

For that, a decision was made to work with the Go programming language since it supplies the tools needed. It provides a way to visualize the Static Single Assignment form (SSA), which is a midway representation that requires each variable to be assigned exactly once and defined before being used. Go compile tool also provides a way of printing some information about the optimizations applied in the compiling process, like whether a variable x is being allocated on the heap or not.

An alternative to the well-known automatic storage methods is the Region-Based Memory Management (RBMM). There are some attempts to implement this kind of techniques. In Grossman et al. (2002), aspects of a new language called Cyclone were described. It is designed to be very close to C, but safe with respect to memory management. To bring this safety to the language they implemented a region-based system. However, Cyclone requires programmers to write some explicit annotations.

In a recent paper, Davis (2015) describes two schemes: an alone implementation of region-based algorithm and a technique which combines that with garbage collection. His solution eliminates the need to write those annotations required by Cyclone. Guyer et al. (2006) also chose the mixed strategy but with a different design. Their approach works by inserting frees to a garbage-collected system.

The main goal of this research is to propose an algorithm which also describes a method that mixes the power of garbage collection with a static analysis that aims to reduce the runtime overhead introduced by the garbage collector (GC). This analysis is called Escape Analysis. It works by detecting the maximum amount of variables that are safe to be allocated in the stack, avoiding the need of invoking the GC.

One must note that this kind of analysis has already been adopted by the designers and maintainers of the Go compiler. Although they did a great job in that, it still has some flaws to be addressed. Having said that, the implementation described in this paper can be thought of as a complement of the Escape Analysis currently running in the building

process of every Go program.

This paper is organized as follows. Section 2 overviews the theory needed for better understanding of what was proposed. Section 3 presents the works related to compile-time strategies for memory management, either fully static or a mix of that with runtime techniques. Section 4 introduces the resources as well as the benchmarks used in this work. Section 5 describes one of those cases not covered yet by the analysis implemented in the Go compiler. Section 6 describes the algorithm implemented. Section 7 presents the results achieved. Finally, section 8 discusses the present and the future of this work.

2. Background

2.1. Static Single Assignment Form

In compiler design, Static Single Assignment Form (SSA) is a property of an intermediate representation, which requires each variable to be assigned exactly once. That is, if a variable *x* was assigned twice in the source code, the translated version will present two distinct values (one for each assignment). Moreover, every variable must be defined before it is used. This kind of representation can facilitate the implementation of a bunch of optimizations, such as constant propagation, register allocation and dead code elimination.

The Go compiler adopt the SSA as its intermediate code, applying all the optimizations on top of that form. A function in the Go's SSA representation mainly consists of a name, a type (its signature), a list of blocks that form its body, and the entry block within said list. When a function is called, the control flow is handed to its entry block. If the function terminates, the control flow will eventually reach an exit block. Furthermore, a function may have zero or multiple exit blocks, since a Go function can have any number of return points, but it must have exactly one entry point block.

A block represents a basic block in the control flow graph of a function. It is, essentially, a list of values that define the operation of this block. It also contains a unique identifier, a kind (e.g. plain, if and exist), and a list of successor blocks. The plain block simply hands the control flow to another block, thus its successors list contains one block. The exit block have a final value, called control value, which must return a memory state. Finally, the if block has a single control value that must be a boolean value, and it has exactly two successor blocks.

A value is the basic building block of SSA. It must be defined exactly once, but it may be used any number of times. Moreover, it mainly consists of a unique identifier, an operator, a type, and some arguments. An operator describes the operation that computes the value. If it takes a memory argument then it depends on that memory state. Besides, an operator where the return type is the memory type impacts the state of memory. This ensures that memory operations are kept in the right order.

For example, the *store* operator take three values as arguments. The first two are the destination and source values, respectively. The last argument is the memory state. At last, the value returned by *store* is of memory type. It may be represented as follows:

$$v_n \; = \; Store \; < mem > \; \{int\} \; v_i \; v_j \; v_m$$

2.2. Escape Analysis

Escape analysis refers to a compile-time approach that simply establishes whether an object can be stack-allocated or not [Blanchet 2003]. It is a technique that determines the set of items that escape a method invocation [Choi et al. 2003]. An object is said to escape from a function or procedure m if its lifetime exceeds the lifetime of m [Blanchet 1999].

To better illustrate that, let $f : Int \rightarrow *Int$ be a function that receives an integer and returns a pointer to some address that holds another integer. Let's also define another function $g : Int \rightarrow Int$ that takes an integer, calls f and returns another integer. What these functions themselves do is not really important, since the focus here is in what is being returned by f and used by g. Both f and g are defined below. Consider $\&y$ as taking the address of y and $*y$ as getting the actual value pointed by y.

$$
\begin{aligned}
&let\ f(x) = \&y \\
&where\ y = x + 1
\end{aligned}
$$

$$
\begin{aligned}
&let\ g(x) = *y \\
&where\ y = f(x)
\end{aligned}
$$

Suppose that the compiler is designed to allocate y in the stack frame of f. Function g calls f and expects a valid address to be returned. But, at the moment that f returns, its stack frame is released. Therefore, address of y, which was allocated in the stack frame of f, is now invalid. This happens because the lifetime of y surpasses the lifetime of f. Hence, y escapes from f.

This kind of analysis allows some optimizations to be performed. The main improvements are: (a) stack-allocating elements that at first would be allocated in the heap and then reclaimed using garbage collection; and (b) reusing objects no longer needed without invoking GC [Park and Goldberg 1992].

3. Related Work

The literature on heap management is extensive. Most of that relies on runtime techniques. Garbage collection already has a vast variety of implementations that perform better or worse accordingly with the environment specifications (e.g. real-time or distributed systems). However, static and mixed approaches have also been studied for decades. This section overviews the related works based on these strategies.

3.1. Static Analysis

Park and Goldberg (1991) describes a method for reducing the time, code and communication overhead of reference counting. They do that by means of an Escape Analysis. Their approach is based on the observation that such overheads can be reduced by avoiding unnecessary reference count updates. This is done by using statically inferred information about the lifetime of each reference.

Choi et al. (2003) presents a framework for Escape Analysis based on a simple program abstraction called the *connection graph*. It captures the relationships among

heap-allocated items and object references. Blanchet (1999, 2003) also describes a framework for Escape Analysis. They aim to reach not only a precise but also a fast analysis. For that they chose to represent the escaping part of an object as an integer. This integer is said to be the context associated with the object.

Kotzmann and Mössenböck (2005) proposes an intraprocedural and interprocedural algorithm for Escape Analysis in the context of dynamic compilation where the compiler has to cope with dynamic class loading and deoptimization. It was implemented for Sun Microsystems' Java HotSpotTM client compiler and operates on an intermediate representation in SSA form. The analysis is used for scalar replacement of fields and synchronization removal, as well as for stack allocation of objects and fixed sized arrays. The results of the interprocedural analysis support the compiler in inlining decisions and allow actual parameters to be allocated on the caller stack.

Guyer et al. (2006) introduces a new memory management method called *free-me compiler analysis*. It merges a static analysis and a garbage collection technique. It works by automatically inserting calls to the free routine at the point an object dies. On average, the free-me analysis deallocates from 32% up to 80% of all items in their benchmarks.

Stadler et al. (2014) presents an algorithm that performs control flow sensitive Partial Escape Analysis in a dynamic Java compiler. It allows Escape Analysis, Scalar Replacement and Lock Elision to be performed on individual branches. They implemented the algorithm on top of Graal, an open-source Java just-in-time compile. In order to analyze the effect of Partial Escape Analysis, they used the DaCapo, ScalaDaCapo and SpecJBB2005 benchmarks. The results were evaluated in terms of run time, number and size of allocations, and number of monitor operations. It performs particularly well in situations with additional levels of abstraction, such as code generated by the Scala compiler. It reduces the amount of allocated memory by up to 58.5%, and improves performance by up to 33%.

3.2. Region-Based Memory Management

Gay and Aiken (1998) shows a detailed comparison of the performance of regions with explicit malloc/free calls and conservative garbage collection. In order to reach that goal they use a set of allocation-intensive benchmark programs. They conclude that their explicit regions are faster than either malloc/free or conservative garbage collection methods, with the speedup sometimes being up to 16%.

Grossman et al. (2002) focus on the aspects of Cyclone's memory management system. It is implemented as a RBMM that tries to bring safety to the language by preventing dangling-pointer dereferences and space leaks. Their work makes the following technical contributions: (a) region subtyping; (b) simple effects; (c) default annotations; and (d) integration of existencial types. They compared the differences between Cyclone and C codes. Results showed that both the overall changes in the program and the number of region annotations are small.

Davis (2015) describes two implementations of a RBMM system. The first one relates only to the compile-time approach; the second combines regions with a garbage collection technique. He chose to work with the Go programming language. The results showed that in some cases his work provides better performance than the standard garbage collector (GC) implemented by Go. However, there are benchmarks that run faster with

the Go approach. Furthermore, the combined technique often requires more memory space.

The implementation being proposed here does not differ that much from the ones described above. It takes into consideration that heap allocations and deallocations are expensive. Therefore, they need to be avoided as much as possible. It is true that garbage collection algorithms have been improved in the last years. However, it is also true that there is no way to completely remove the runtime overhead introduced by this kind of technique. Moreover, it must be note that each minimal speedup may have impact not only in the programs themselves but also in the resources being used (e.g. energy).

Having said that, the main difference between the approaches above and the one detailed here is that the latter tries to complement an already existing system to attack the flaws not covered yet. By doing that, it is possible to reduce even more the heap allocations introduced by the compiler. Hence, the objective is to make the GC be called only to clear the mess added by objects that cannot have their lifetime tracked at all.

The algorithm being proposed works by building and traversing a directed graph G. Each edge of the set $E \in G$ connects a value v of a function with the values that contains v as an argument (i.e. places where v is being used). Besides that, for each one of those values a region is assigned. That region may be either *inside* or *outside* the function. This is used to determine the value's lifetime.

Furthermore, we walk through the graph G in a depth-first search approach, labeling each node with one of the following states: (a) *safe*; (b) *may escape*; and (c) *must escape*. If a node is said to be *safe*, there is no reason to analyze its adjacent vertex. In contrast, if a node is defined as *must escape* then the analysis is finished and the value being verified indeed escape to the heap.

4. Methodology

4.1. Tools

Go was the language of choice for the development of this project. It gives us the ability to visualize the transformations performed by the compiler. Each of them is called a *pass*. Figure 1 summarize these steps. First, an Abstract Syntax Tree (AST) is generated from the source code. The *start* pass is the Static Single Assignment form (SSA) that is produced from the AST. Listings 1 and 2 show a simple code example written in Go and the SSA generated from it. Intermediate passes do all kinds of optimizations. For the sake of simplicity they were grouped together as *opts*. The *lower* pass converts the SSA representation from being machine-independent to being machine-dependent. Finally, the *genssa* is the final code generated by the compiling process.

Listing 1. Example of source code

```
1 package main
2
3 func foo() {
4     x := 1
5
6     s := make([]*int, 1)
7     s[0] = &x
8 }
```

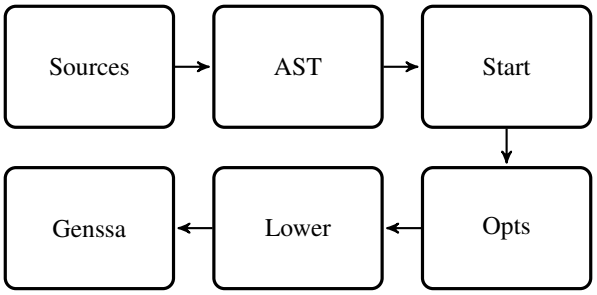

Figure 1. A summary of the transformations executed by the compiler

Listing 2. The SSA generated

```
b1:                                          // A new block label.

    v1 = InitMem <mem>                       // Initialize the memory state.

    v2 = SP <uintptr>                        // Stack Pointer.

    v3 = SB <uintptr>                        // Static base pointer (global
                                             // pointers).

    v4 = OffPtr <**byte> [0] v2              // Pointer to the stack with
                                             // offset 0.

    v5 = Addr <*uint8> {type.[2]int} v3      // Address of a pointer to the
                                             // type [2]int. This is the
                                             // type of the value that will
                                             // be allocated in the heap.
                                             // The returned value (the
                                             // address) is *uint8, which is
                                             // the same as a *byte.

    v6 = Store <mem> {*byte} v4 v5 v1        // Store v5 (*byte) to v4
                                             // (**byte). The last arg (v1)
                                             // is the memory state (i.e.
                                             // the value related to the
                                             // last operation that changed
                                             // the memory state). Store
                                             // operation changes the memory
                                             // state since its return type
                                             // is "mem".

    v7 = StaticCall <mem> {runtime.newobject} [16] v6
                                             // Call to the runtime function
                                             // "newobject" that is
                                             // responsible to allocate a
                                             // new value in the heap. As
                                             // Store, StaticCall operation
                                             // also changes the memory
                                             // state.

    v8 = OffPtr <**int> [8] v2               // Pointer to the stack with
                                             // offset 0.

```

```
43      v9 = Load <*int> v8 v7 (&x[*int])    // Get the value (address)
44                                           // that will be returned by
45                                           // runtime.newobject.
46      ⋮
```

The Go compiler also allows one to print optimization decisions by passing the flag *-m* to the compile command. Assuming the code illustrated in listing 1 is placed inside a file named *main.go*, when running the command *go tool compile -m main.go* the output is the following:

$$
\begin{aligned}
&main.go : 3 : 6 : \ can\ inline\ foo \\
&main.go : 7 : 9 : \ \&x\ escapes\ to\ heap \\
&main.go : 4 : 2 : \ moved\ to\ heap : \ x \\
&main.go : 6 : 11 : \ foo\ make([] * int,\ 1)\ does\ not\ escape
\end{aligned}
$$

The second and third lines of the output show that the lifetime of the variable x surpasses the lifetime of the function foo. Therefore, x needs to be heap-allocated. The Go compiler already has an escape analysis implemented, but still there are some cases not covered yet. We described one of them later in section 5. Thereby, the present work aims to address these situations.

4.2. Benchmarks

In order to analyze the results of this work a total of five benchmarks were used, split into one synthetic and four real applications. The synthetic one (listing 3) extends the example presented on section 5 by appending elements to a slice. The total number of insertions follows the slice size, which varies from 128 up to 1024.

Listing 3. Synthetic benchmark

```
1  package main
2
3  import (
4      "math/rand"
5      "testing"
6  )
7
8  const (
9      size      = 128 // 128, 256, 512 and 1024.
10     dummyVal = 10000
11 )
12
13 // BenchmarkSlice ...
14 func BenchmarkSlice(b *testing.B) {
15     for i := 0; i < b.N; i++ {
16         slice := make([]*int, size)
17
18         slice[0] = new(int)
19         *slice[0] = dummyVal
```

```
20
21              slice[1] = new(int)
22              *slice[1] = dummyVal
23
24              ...
25
26              slice[size-1] = new(int)
27              *slice[size-1] = dummyVal
28      }
29  }
```

The remaining programs are real benchmarks maintained in the Go's official repository[2]. Table 1 presents a small description for the real programs of the benchmark suite. Furthermore, these applications were tested with respect to their performance (i.e. response time). To properly process the results achieved we used the tool *benchstat*[3]. Finally, to ensure the maximum consistency of the data collected, an average of the response time between a hundred executions was taken for each benchmark in the suite.

Benchmark	Description
Build	Examines compiler and linker performance
Garbage	Stresses the garbage collector
Http	Examines client/server http performance
Json	Marshals and unmarshals approximately 2MB json string with a tree-like object hierarchy

Table 1. Benchmark descriptions

5. Motivation

Despite the existing implementation of Escape Analysis being already an efficient one, there are still some flaws. Listing 4 illustrates a situation in which it is possible to guarantee the safety of stack-allocating some value but it is still escaping to the heap. Although it may be possible to address these cases by extending the current algorithm, the power of the SSA form ends up offering some facilities to track the lifetime of x. For example, one may easily build a graph with all direct and indirect references to an element. This is the main reason behind the option of developing an algorithm from scratch.

Consider the function *foo* showed in the listing 4. It creates a slice s of pointers to integers and assigns the address of the variable x to the first position of s. It also constructs a map where the key is an integer and the value is a pointer to an integer. Similarly to the slice case, *foo* sets the element related to the key *0* as the address of the variable y.

It is clear that there is no reason for moving the values under the variables x and y to the heap, since their lifetime is the same as of the function where they were created. Still, what happens is that the slice and the map themselves do not escape, but any value inserted into such structure does. Therefore, the elements related to x and y are moved to the heap. The output of the compilation process of this function with the flag *-m* follows:

[2]https://github.com/golang/benchmarks
[3]https://godoc.org/golang.org/x/perf/cmd/benchstat

$$main.go : 4 : 2 : \ moved\ to\ heap : \ x$$
$$main.go : 8 : 2 : \ moved\ to\ heap : \ y$$
$$main.go : 5 : 11 : \ foo\ make([] * int,\ 1) does\ not\ escape$$
$$main.go : 9 : 11 : \ foo\ make(map[int] * int,\ 1)\ does\ not\ escape$$

Listing 4. Assigment to a slice and a map

```
1  package main
2
3  func foo() {
4      x := 1
5      s := make([]*int, 1)
6      s[0] = &x
7
8      y := 1
9      m := make(map[int]*int, 1)
10     m[0] = &y
11 }
```

6. Proposed Algorithm

Let $G = \langle V, E \rangle$ be a directed graph that connects a node $v \in V$ (in the SSA form) with each reference (direct or indirect) to it. A value v is said to be referenced by $u \in V$ if v appears as one of the arguments of u. Each vertex can be labeled as one of the following states: *safe*, *may escape* or *must escape*. Moreover, if there is a path from v to u and u is categorized as *must escape*, then v must also be defined as *must escape*. Figure 2 depicts the graph constructed from the values of the listing 2 as an example.

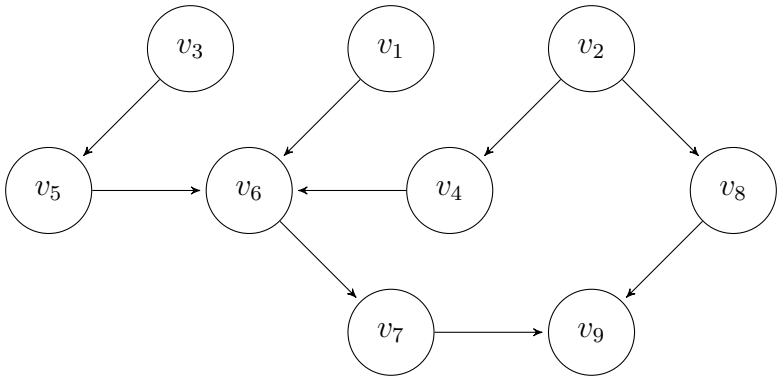

Figure 2. Example of a directed graph built from the values in the SSA form

The first step is to build the graph G. This is accomplished by calling a method $BuildGraph$ (algorithm 1) which runs through all the values of a given function F. Furthermore, during this step the algorithm also determines whether the lifetime of a value surpasses or not the lifetime of the function being analyzed. For that each node will be assigned to a region by calling the method $GetRegion$ (algorithm 2).

This region can be either inside or outside of F. If the region of a node v is said to be *outside* and a node u can be reached from v, then the region of u must also be set

as *outside*. Hence, we also define a function named $PropagRegion$ (algorithm 3) which is responsible to propagate the region of each value that is considered as being outside of F. This is accomplished by means of a breadth-first search approach.

The core algorithm works by traversing G in a depth-first search manner. This is done by a function named $Walk$ (algorithm 4) that takes the graph G and a start node s as parameters. The initial vertex s is the one related to the value currently being heap-allocated and which is going to be analyzed in order to determine if it can be instead stack-allocated. For each node u reached, a function $Visit$ is applied to verify if u must escape or not. This function is detailed in algorithm 5.

If the result of applying $Visit$ to a value u is *must escape*, then s is also defined as *must escape*. Hence, there is no need to visit any other vertex. Else, if the result is *safe*, then the algorithm prunes the graph by not visiting the edges that start from u. Otherwise, the algorithm must continue traversing G (i.e. the *may escape* state is only a intermediate one).

Algorithm 1 Creates a new graph G connecting each value v with its references u_i

```
function BUILDGRAPH(F)
    let G be a new graph
    for each block b ∈ F do
        for each value v ∈ b do
            call GetRegion passing v to it
            set the region of v as the region returned by GetRegion
            add v as a node of G

    for each block b ∈ F do
        for each value v ∈ b do
            for each argument a ∈ v do
                add an edge from a to v

    return G
```

Algorithm 2 Takes a value v and returns the corresponding region

```
function GETREGION(v)
    if v is global then
        return outside
    else if v is a param (in/out) and the type t of v holds (or is) a pointer then
        return outside
    else if type t of v contains any underlying value (e.g. pointer or slice) then
        loop through the underlying element until reaching the final type f
        if f contains a pointer to the heap then
            return outside
    else
        return inside
```

Algorithm 3 Walks through G in a BFS approach and propagates the region r of a value u to other nodes w_i if r is $oustide$

```
function PROPAGREGION(G)
    for each node v ∈ G do
        if region r of v is inside then skip to the next iteration
        let Q be a new queue
        enqueue v at Q

        while Q ≠ ∅ do
            dequeue u from Q
            for each adjacent node w of u not visited yet do
                set the region r' of w to outside
                enqueue w at Q
                mark w as visited
```

7. Results

We implemented a Escape Analysis on top of the SSA intermediate form as one of the many passes done by the Go compiler. This decision was primarily made because of the flexibility of building the analysis from scratch and without dependencies on any other step of the compiling process. In order to evaluate the algorithm described in this paper, we used four real benchmarks and one synthetic program. The response times R_i^{old} and R_i^{new} represents the run time of the original version and the one with the proposed algorithm implemented. The speedup S_i of a benchmark i is obtained as follows:

$$S_i = \frac{R_i^{old}}{R_i^{new}}$$

7.1. Synthetic Benchmark

Figure 3 presents a comparison between the run time of the synthetic benchmark when compiled with and without the escape analysis proposed. We tested the synthetic program configuring the slice size as 128, 256, 512 and 1024. Figure 4 shows the growth of the response time when increasing the slice size. Lastly, table 2 contains the run time for each test along with their respective speedups.

The results collected from the synthetic benchmark evidence the potential related to the escape analysis described in this paper. It is clear that the overhead introduced by the management of the heap area is high. Not only that, but when dealing with data structures like lists this overhead also increases according to the number of elements being manipulated.

7.2. Real Benchmarks

Figure 5 presents a visual representation of the data collected from the benchmarks. Table 3 shows the average response time obtained for each application together with their speedups. The response times related to the build benchmark are the same for both the original and new versions, i.e. no speedup was reached. Nonetheless, the remaining benchmarks indeed presented improvements when compiled considering the proposed

Algorithm 4 Walks through G in a DFS approach and applies $Visit$ to each node reached

```
function WALK(G, s)
    let S be a new stack
    push the parents u_i of v to S

    while S ≠ ∅ do
        let u be the value popped from S
        if u was already visited then
            skip to the next iteration

        Visit(u)
        mark u as visited

        if state of u is must escape then
            set the state of s as must escape
            break the while loop

        if state of u is safe then
            set the state of s as safe
            skip to the next iteration

        push the parents t_i of u to S
```

Size	Old response time (σ)	New response time (σ)	Speedup
128	1.95 μs (0.002 μs)	0.04 μs (0.00004 μs)	48.75
256	3.98 μs (0.012 μs)	0.08 μs (0.00006 μs)	49.75
512	8.01 μs (0.030 μs)	0.18 μs (0.00040 μs)	44.50
1024	16.1 μs (0.027 μs)	0.40 μs (0.00020 μs)	40.25

Table 2. Speedups related to each slice size

algorithm. The json application was the one that achieved the biggest speedup (approximately 3.4%).

By analyzing the experimental results the initial conclusion is that the implementation did not perform well when verified against a single program execution. However, this kind of optimization not only impacts a single program but several of them. In this sense, the response times accomplished were significant. Moreover, the algorithm developed was also able to successfully target one of the flaws (detailed in section 5) of the existing escape analysis implementation.

8. Conclusion and Future Works

We presented a type of static analysis responsible for identifying situations where it is safe to store in the stack a value that would initially be heap-allocated. The results show that the algorithm have high potential to reduce the overhead caused by the GC and, consequently, the response times of the applications. The algorithm was also capable of

Algorithm 5 Visit a node v to analyze if this value should escape to the heap

```
function VISIT(v)
    if v does not holds or isn't a pointer then
        set the state of v as safe
    else if v is related to a operation with write semantics then
        if the source value does not contains any pointer then
            set the state of v as safe
        else if the region of the destiny value is outside then
            set the state of v as must escape
        else
            set the state of v as may escape
    else
        set the state of v as may escape
```

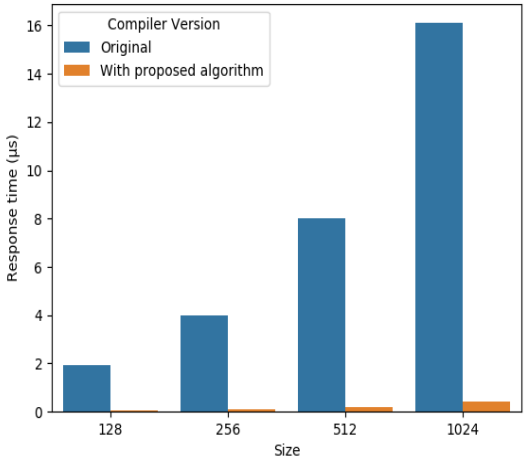

Figure 3. Response time per size of the slice

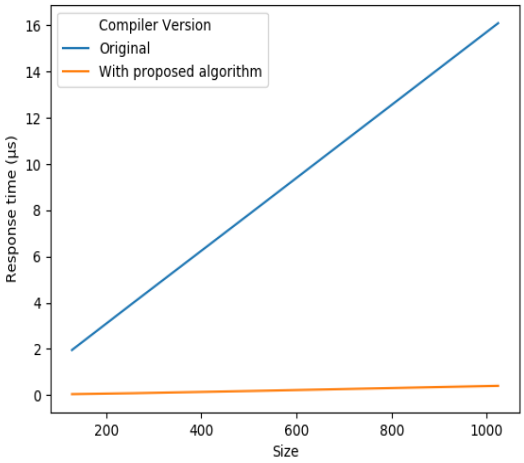

Figure 4. Response Time Growth

targeting the case described in section 5, meaning that there is at least one situation that could be improved in not one but all of the Go programs that it may appears.

The speedups obtained from the real benchmarks were limited due to some restrictions faced during the development of this project. The most important among them was the one related to the maximum depth of the analysis. In other words, we could not find a way of verifying the functions called by the current one being verified. Thereby, if a value $v \in G'$ (where G' is the subgraph related to the heap-allocated element u being analyzed) was being passed to any function as a parameter, then we had to give up and consider that u escapes for sure. Hence, there is space to improve even more the current implementation.

This improvement may be reached in some distinct ways. At first, one could delve into the compiler code looking for approaches to enter into functions called by the current one being analyzed. This way it would be possible to eliminate the restriction described before, resulting in a deeper analysis. However, taking this path may probably lead into

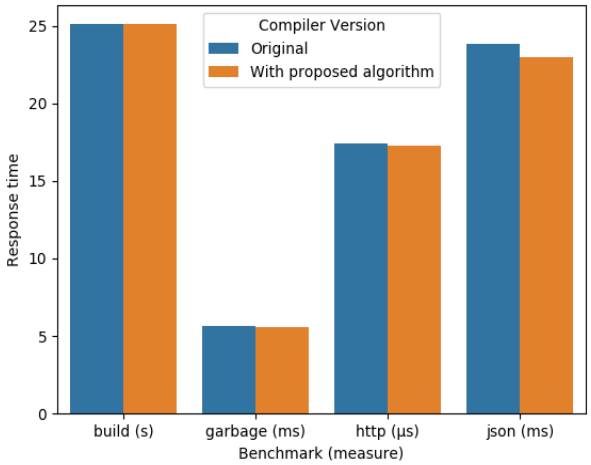

Figure 5. Response time for each benchmark

Benchmark	Old response time (σ)	New response time (σ)	Speedup
Build	25.1 s (0.13 s)	25.1 s (1.15 s)	1
Garbage	5.67 ms (0.11 ms)	5.59 ms (0.09 ms)	1.014
Http	17.4 μs (0.06 μs)	17.3 μs (0.07 μs)	1.005
Json	23.8 ms (0.93 ms)	23 ms (0.58 ms)	1.034

Table 3. Benchmark speedups

much higher compilation times.

A second option would be to insert calls to a deallocating routine related to the garbage collector instead of rewriting the allocation instructions. While this method cannot eliminate all of the overhead introduced by the garbage collector, it could cover a lot of more cases than the rewriting approach.

To illustrate, consider the synthetic benchmark rewritten as a loop. With the algorithm proposed here, there is no way to transform the heap-allocation since the stack address will always be the same. This happens because the lifetime of the element will be related to the loop, i.e. the value is created and destroyed in that block. In such case, introducing a call to some function responsible to destroy that object instead of rewriting the allocation may be a more efficient proposal.

References

Barth, J. (1977). Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518.

Blanchet, B. (1999). Escape analysis for object-oriented languages: Application to java. *SIGPLAN Not.*, 34(10):20–34.

Blanchet, B. (2003). Escape analysis for Java (TM): Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775.

Choi, J., Gupta, M., Serrano, M., Sreedhar, V., and Midkiff, S. (2003). Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):876–910.

Davis, M. (2015). *Automatic memory management techniques for the go programming language*. PhD thesis, Department of Computer Science and Software Engineering The University of Melbourne. Retrieved from `http://hdl.handle.net/11343/58707`.

Gay, D. and Aiken, A. (1998). Memory management with explicit regions. *ACM SIGPLAN NOTICES*, 33(5):313–323. Annual SIGPLAN Conference on Programming Language Design and Implementation, Montreal, Canada, Jun 16-19, 1998.

Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., and Cheney, J. (2002). Region-based memory management in cyclone. *ACM SIGPLAN Notices*, 37(5):282–293. Conference on Programming Language Design and Implementation (PLDI 02), BERLIN, GERMANY, JUN 17-19, 2002.

Guyer, S. Z., McKinley, K. S., and Frampton, D. (2006). Free-me: A static analysis for automatic individual object reclamation. *ACM SIGPLAN Notices*, 41(6):364–375.

Iyengar, A. (1996). Scalability of dynamic storage allocation algorithms. In *Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96)*, pages 223–232.

Kotzmann, T. and Mössenböck, H. (2005). Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 111–120, New York, NY, USA. ACM.

Louden, K. C. (1997). *Compiler Construction: Principles and Practice*. PWS Publishing Co., Boston, MA, USA.

Park, Y. and Goldberg, B. (1992). Escape Analysis on Lists. *SIGPLAN Notices*, 27(7):116–127.

Park, Y. G. and Goldberg, B. (1991). Reference escape analysis: Optimizing reference counting based on the lifetime of references. In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '91, pages 178–189, New York, NY, USA. ACM.

Stadler, L., Würthinger, T., and Mössenböck, H. (2014). Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 165:165–165:174, New York, NY, USA. ACM.