Escape from Escape Analysis of Golang

Cong Wang
KLISS, BNRist, School of Software,
Tsinghua University
Beijing, China

Mingrui Zhang
KLISS, BNRist, School of Software,
Tsinghua University
Beijing, China

Yu Jiang*
KLISS, BNRist, School of Software,
Tsinghua University
Beijing, China

Huafeng Zhang
Compiler and Programming
Language Lab, Huawei Technologies
Hangzhou, China

Zhenchang Xing
College of Engineering and Computer
Science, ANU
Canberra, Australia

Ming Gu
KLISS, BNRist, School of Software,
Tsinghua University
Beijing, China

ABSTRACT

Escape analysis is widely used to determine the scope of variables, and is an effective way to optimize memory usage. However, the escape analysis algorithm can hardly reach 100% accurate, mistakes of which can lead to a waste of heap memory. It is challenging to ensure the correctness of programs for memory optimization.

In this paper, we propose an escape analysis optimization approach for Go programming language (Golang), aiming to save heap memory usage of programs. First, we compile the source code to capture information of escaped variables. Then, we change the code so that some of these variables can bypass Golang's escape analysis mechanism, thereby saving heap memory usage and reducing the pressure of memory garbage collection. Next, we present a verification method to validate the correctness of programs, and evaluate the effect of memory optimization. We implement the approach to an automatic tool and make it open-source[1]. For evaluation, we apply our approach to 10 open-source projects. For the optimized Golang code, the heap allocation is reduced by 8.88% in average, and the heap usage is reduced by 8.78% in average. Time consumption is reduced by 9.48% in average, while the cumulative time of GC pause is reduced by 5.64% in average. We also apply our approach to 16 industrial projects in Bytedance Technology. Our approach successfully finds 452 optimized cases which are confirmed by developers.

CCS CONCEPTS

• Software and its engineering → General programming languages; **Compilers**.

KEYWORDS

escape analysis, memory optimization, code generation, go programming language

ACM Reference Format:
Cong Wang, Mingrui Zhang, Yu Jiang, Huafeng Zhang, Zhenchang Xing, and Ming Gu. 2020. Escape from Escape Analysis of Golang. In *Software Engineering in Practice (ICSE-SEIP '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3377813.3381368

1 INTRODUCTION

Memory management is very important for software engineering. According to the statistics of Bytedance Technology [2], there were 114 memory-related online accidents in 2018. For example, cloud services are running in containers which have limits of memory usage. Some of these accidents occur when the memory cost exceeds the limit. Today, more and more internet companies are concerned about memory management and optimization issues.

Escape analysis [7, 11, 15, 22] is an algorithm to determine the scope of pointers, and is an effective way to optimize memory usage. The escape analysis mechanism determines whether to save a variable in heap. At any time, a value is reassigned on the heap if it is shared outside the scope of the function stack. Escape analysis discovers these conditions in the compile phase. Although escape analysis is very helpful for memory optimization [4, 24], its algorithm could hardly be completely correct [7]. If a large object is moved to heap memory but is not being accessed externally, it leads to a waste of heap memory. For example, Listing. 1 shows a Golang (short for Go programming language) program. In this program, a pointer (addrObj) of variable "obj" is used as a function parameter (Line 19). Then the escape analysis algorithm determines that "obj" should be moved to the heap. However, in this program, "obj" is no longer shared by other functions. This could be a waste of heap memory, especially when "obj" is a large object. And the situation will trigger unnecessary garbage collection (GC) [6, 17, 25] and subsequently affect program performance. From the above situation, we hope to optimize programs for such cases and bypass the escape analysis to avoid wasting memory.

```
1 // file: escape.go
2 package main
3 import "fmt"
```

*Yu Jiang is the correspondence author. This research is sponsored in part by National Key Research and Development Project (Grant No. 2019YFB1706200, No. 2016QY07X1402, 61400010107) the NSFC Program (No. 61802223), and the Equipment Pre-research Project (No. 61400010107).

[1] Tool Link: https://github.com/wangcong15/escape-from-escape-analysis-of-golang

[2] Bytedance is a outstanding Chinese software company which develops mobile applications, such as Tik Tok, TopBuzz, News Republic, etc.

ICSE-SEIP '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7123-0/20/05…$15.00
https://doi.org/10.1145/3377813.3381368

```
type BO struct {
    field1 []int
}
func causeEscape(i interface{}) {
    switch i.(type) {
    case *BO:
        println(i)
    default:
        fmt.Println(i)
    }
}
func main() {
    obj := BO{}
    obj.field1 = make([]int, 5000)
    addrObj := &obj
    causeEscape(addrObj)
}
```

Listing 1: A Golang Program: Variable "obj" is moved to heap by escape analysis mechanism.

There has been an amount of work on escape analysis in the context of Java [4, 5, 7, 8, 24]. Meanwhile, escape analysis techniques have also been applied in functional languages [3, 9, 13, 14, 19], and multi-threaded programs [16, 21]. Compared to traditional and mature programming languages, in Golang, the code grammatical structure has changed a lot (such as channel, interface, etc). The escape analysis algorithm is also different. As written in Golang's official documents [12], there are many compromises in Golang's escape analysis algorithm, and there could be a number of variables stored in heap which leads to a waste of memory in real practice.

Memory optimization on escape analysis is helpful to reduce the excessive use of heap memory, to reduce the frequency of garbage collection, and to improve the efficiency of code execution. In this paper, we will focus on the memory optimization of escape analysis for Golang. However, this task is challenging from two major perspectives.

- *Optimization.* Our approach should be able to optimize Golang programs to bypass Golang's escape analysis mechanism and reduce memory usage.
- *Memory Integrity.* The optimization task requires that the optimized program does not have problems with memory readings and writings [1]. Execution of optimized Golang programs should not crash or change.

In this paper, we propose an escape analysis optimization approach for Golang, aiming to save heap memory usage of programs. First, we compile the source code to capture information of escaped variables. Then, we optimize the code so that these variables can bypass Golang's escape analysis mechanism, thereby saving heap memory usage and reducing the pressure of memory garbage collection. Next, we present a verification method to validate the correctness of programs, and evaluate the effect of memory optimization. For evaluation, we apply our approach to 10 open-source projects. For the optimized Golang code, the heap allocation is reduced by 8.88% in average, and the heap usage is reduced by 8.78% in average. Time consumption is reduced by 9.48% in average, while the cumulative time of GC pause is reduced by 5.64% in average. We also apply our approach to 16 industrial projects in Bytedance Technology. Our approach successfully finds 452 optimized cases which are confirmed by developers. The experimental results prove the correctness and effectiveness. The main contributions are:

- We propose an escape analysis optimization approach for Golang. Based on the approach, we can save heap memory usage of programs.
- We present a prototype implementation on our approach, which can optimize memory usage in practice.
- We evaluate the performance on open-source and industrial programs. The experimental results show that our approach is effective in memory optimization.

The rest of this paper is organized as follows. Section.2 describes the related work and main differences. Section.3 elaborates on the approach of escape analysis optimization, includes the escape capture, code optimization, and correctness verification. Section.4 presents experimental results. Section.5 presents the lessons learned from the practice and we conclude in Section.6.

2 RELATED WORK

Escape Analysis. There has been an amount of work on escape analysis [10, 16, 18, 20, 21, 23]. Choi et al. [7] propose a framework of escape analysis and demonstrate an application on Java programs. They present an interprocedural algorithm to efficiently compute the connection graph and identify the non-escaping objects for methods and threads. Meanwhile, escape analysis techniques have been applied in functional languages [3, 9, 19]. Deutsch [9] presents a static method for determining aliasing and lifetime of dynamically allocated data in functional specifications. His approach is based on an operational model of higher-order functional programs from which we construct statically computable abstractions using the abstract interpretation framework. Compared to traditional and mature programming languages, in Golang, the escape analysis algorithm is much simpler. For example, expressions assigned to any kind of indirection (*p=...) are considered escaped. This issue happens because Golang's compiler would rather waste memory than make code crash. As written in Golang's official document [12], there are many compromises in Golang's escape analysis algorithm, which means that in practice there could be a number of variables stored in heap memory when they are not shared. Other things that can inhibit analysis are function calls, package boundaries, slice literals, subslicing and indexing, etc.

Multi-threaded Escape Analysis. Escape analysis is also used in multi-threaded programs [16, 21]. Yulei et al. [21] propose the sparse flow-sensitive pointer analysis for unstructured multi-threaded programs (C with Pthread). Their method is significantly faster than non-sparse algorithms. Jeff Huang [16] presents two algorithms (a static algorithm and a dynamic one) for identifying program statements that access thread-shared data in concurrent programs. His results suggest that the two algorithms are promising for practical use in analyzing real-world large-scale multi-threaded programs.

Our Differences. We do optimization for an escape analysis situation, in which a pointer variable is used as a function argument. When this situation happens, the Golang compiler decides to move the pointed object to heap memory. Sometimes this choice leads to a waste of memory. Notes that this situation has been addressed in some other programming languages, such as Java. Our work is not to design an escape analysis algorithm, but to present an optimization practice on Golang's existing escape analysis mechanism. We

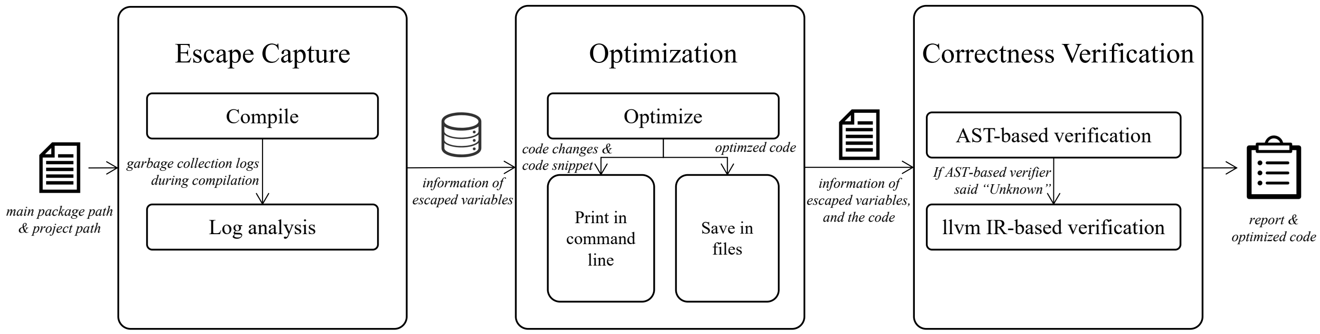

Figure 1: Overall Framework: escape capture phase, optimization phase and correctness verification phase.

hope to perform optimization in order to bypass escape analysis and avoid wasting heap memory. This optimization meets some challenges, including memory integrity problems, optimization effectiveness, and scalability with multi-threaded programming. Therefore, we try to achieve the optimization in code level and implement the approach as an automatic tool. When optimizing memory for escape analysis, we guarantee the memory integrity via a verification method. More importantly, our approach successfully reduces the waste of heap memory and releases the pressure of garbage collection.

3 APPROACH

Firstly we introduce the overall framework in Section.3.1. We use an example to illustrate the approach overview. Then details of modules are described in the following sub-sections.

3.1 Overview

As shown in Figure. 1, our approach consists of three main phases: escape capture phase, optimization phase and correctness verification phase. First, the approach requires Golang programs as input. In "Escape Capture" phase, we compile the source code and analyze log records of garbage collection (GC). Then in "Optimization" phase, we optimize the source code. We print the changed code snippets and save the optimized code into files. Finally, in "Correctness Verification" phase, we verify the correctness of memory integrity by AST-based and llvm IR-based verification methods.

We use Listing. 1 as an example to illustrate the approach. First we get garbage collection (GC) logs by compiling source code. According to GC logs, escape variables are show in Table. 1. Column "Variable" lists four variables which are moved to heap memory during compilation. Column "Traces" explains the reasons why a variable escapes. For example, the first variable "addrObj" escapes because it is used as a function argument at ./main.go (Line.19, Offset.13). In this example, variable "obj" is relatively large, which contains a field as an integer array (field1). Then, we will change the code, print the snippet in terminal and save it to file. In this case, the strategy of optimization is to make a smaller variable escape instead. In Golang, "uintptr" is an integer type that is large enough to hold the bit pattern of any pointer in Golang. Type conversion between

Variable	Traces
addrObj	from addrObj, passed to call, at ./main.go:19:13
&obj	from addrObj, assigned, at ./main.go:19:13 from addrObj, interface-converted, at ./main.go:19:13 from addrObj, passed to call, at ./main.go:19:13
make	from obj, dot-equals, at ./main.go:17:13 from &obj, address-of, at ./main.go:18:13 from addrObj, assigned, at ./main.go:18:10 from addrObj, interface-converted, at ./main.go:19:13 from addrObj, passed to call, at ./main.go:19:13
obj	&obj escapes to heap

Table 1: Escape Variables in Listing. 1

pointer and uintptr could bypass compiler's escape analysis algorithm. The optimized code of Listing. 1 is shown in Listing. 2. The Golang compiler does not find a connection between obj (Line.17) and i (Line.8). Therefore, compared to the original code in Listing.1, only "(*BO)(unsafe.Pointer(addr))" escape to heap memory. That takes up only the size of a pointer in heap memory.

```
1  // file: non-escape.go
2  package main
3  import "fmt"
4  import "unsafe"
5  type BO struct {
6      field1 []int
7  }
8  func causeEscape(i interface{}) {
9      switch i.(type) {
10     case *BO:
11         println(i)
12     default:
13         fmt.Println(i)
14     }
15 }
16 func main() {
17     obj := BO{}
18     obj.field1 = make([]int, 5000)
19     addrObj := &obj
20     addr := uintptr(unsafe.Pointer(addrObj))
21     causeEscape((*BO)(unsafe.Pointer(addr)))
22 }
```

Listing 2: Opimized Code.

As shown in Figure. 2, we make a comparison of memory usage for these two code snippets. The two programs take up some heap memory. In this figure, "HeapAlloc" is bytes of allocated heap objects. "HeapInUse" is bytes in in-use spans. From the comparison of data, we can draw a preliminary conclusion that the use of heap memory is reduced through this optimization algorithm. This is only a tiny example to demonstrate. We will present the optimization performance of our approach in Section.4. Then, we need to verify the correctness to ensure the memory integrity. We design a static verification method. The verification method analyze the control flow and data flow to ensure the correctness and consistence of the optimized programs.

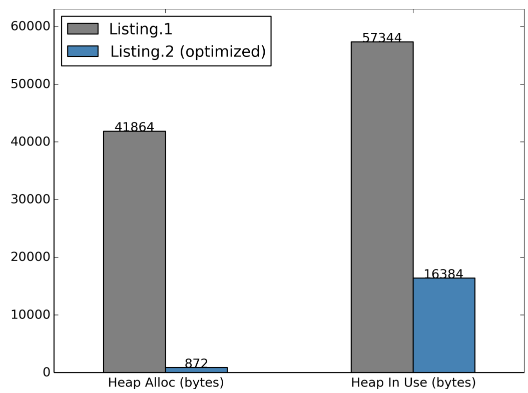

Figure 2: Comparison on Heap Memory Usage

3.2 Escape Capture Phase

In this paper, we do optimization for an escape analysis situation, in which a pointer variable is used as a function argument. When this situation happens, the Golang compiler decides to move the pointed object to heap memory. Sometimes this choice leads to a waste of memory. "Escape Capture" phase aims to compile a Golang project and capture information of escaped variables of the above situation. Here, we denote a model **Context** to record run-time information. Context contains attributes as follows:

- *ArgMain* is a user-provided argument which specifies the main Golang package.
- *ArgPkg* is a user-provided argument which specifies the Golang project. We will deal with some of the code, which are related to the main package.
- *PathToLog* is a hash map (a.k.a "dictionary" in some programming languages). Keys are file paths, while values are compilation logs. *Context.PathToLog* records the compliation logs for each file.
- *EscapeCases* is also a hash map. Keys are file paths. Values are lists of escaped variables, including pointer variable name (PtrName), line number (LineNo) and the verification result (IsCorrect).

Details of the escape capture are presented in Algorithm. 1. Usually, an executable Golang project contains a main package. $Path_m$ denotes the path of this main package. $Path_p$ is the path of the

Algorithm 1 Escape Capture

Input:
Path of the main package, $Path_m$
Path of the project, $Path_p$
Output:
Context to record run-time information, ctx

1: $ctx \Leftarrow$ a new Context structure
2: $ctx.ArgMain, ctx.ArgPkg = Path_m, Path_p$
3: $ctx.PathToLog[ctx.ArgMain] = getGcLog(ctx.ArgMain)$
4: **for** each $pkg \in getDeps(ctx.ArgMain, ctx.ArgPkg)$ **do**
5: $\quad ctx.PathToLog[pkg] = getGcLog(pkg)$
6: **end for**
7: $rM \Leftarrow$ a regular expression for object moved to heap
8: $rI \Leftarrow$ a regular expression for interface conversion
9: $rP \Leftarrow$ a regular expression for pass to function call
10: **for** each $pkg \in ctx.PathToLog$ **do**
11: $\quad$ **if** rM, rI, rP match variable v in $ctx.PathToLog[pkg]$ **then**
12: $\quad\quad ec \Leftarrow$ fetch v's messages as $(PtrName, LineNo)$
13: $\quad\quad ctx.EscapeCases[pkg] \Leftarrow ctx.EscapeCases[pkg] \cup ec$
14: $\quad$ **end if**
15: **end for**
16: **return** ctx

Golang project and determines the scope of the optimization procedure. The output of the escape capture algorithm is a Context object. In Line.1, ctx is assigned as a new Context object. In Line.2, ctx records the user-provided arguments. Then in Line.3, We record terminal logs of compilation. Here, the method "getGcLog" is to execute a command[3], and fetch the terminal output. This terminal output contains logs of escaped variables, including the reasons why they are moved to heap. Then we will use this information to pick some escaped variables which is caused by function arguments. In Line.4-6, we record terminal logs of compilation for each related Golang packages. Till this step, raw logs are ready. Line.7-9 initiate three regular expression handlers: rM, rI and rP. When scanning the logs, we use the three regular expressions to find matched escape cases. In Line.10-15, we repeat this operation for each Golang package. When regular expression check passes, we fetch the messages of escaped variable and record them in $ctx.EscapeCases$. Finally ctx is returned back.

3.3 Optimization Phase

In the escape capture phase, we compile the Golang source code and capture the terminal output. Notes that in this paper, we focus on the situation, in which a pointer variable is used as a function argument. In other words, the escape capture algorithm successfully saves the information of escaped variables for the situation. These messages are saved in $ctx.EscapeCases$. In the optimization phase, we will optimize code to bypass Golang's escape analysis algorithm, and to optimize the heap memory usage.

Figure. 3 shows the basic principle of optimization. As shown in Figure. 3.(A), the original code pass the normal pointer to function call. This situation is considered to be an escape. Our method is to do transformation between normal pointer, unsafe.Pointer and uintptr,

[3]The command is: cd DIR && go build -gcflags="-m -m" *.go

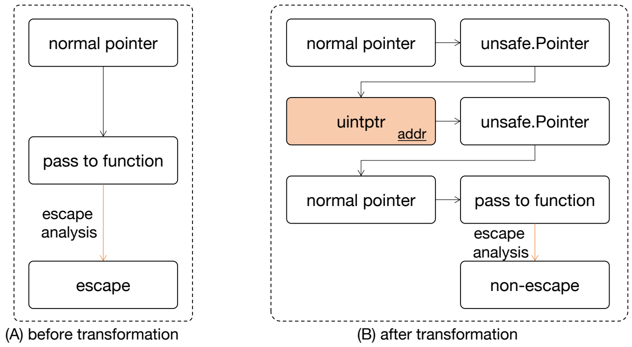

Figure 3: Code Optimization Preview

as shown in Figure. 3.(B). Uintptr is an integer type that is large enough to hold the bit pattern of any pointer. Unsafe.Pointer is a pointer type defined in Golang. Then we rewrite the optimized code into source files. Golang compiler no longer considers the normal pointer to have a referral relationship with the function parameter, thus bypassing the inner escape analysis algorithm. Listing. 3 shows an example of optimization procedure. The procedure is simple and effective. The old function call is removed as shown in Line.5. Then the optimization procedure adds two new code statements in Line.6-7. Line.6 declares a new "uintptr" variable, which is used to by-pass Golang's escape analysis mechanism. Line.7 calls the original function by using "uintptr" instead of the original pointer variable "addrObj".

```
1 func main() {
2     obj := BO{}
3     obj.field1 = make([]int, 5000)
4     addrObj := &obj
5 --  causeEscape(addrObj)
6 ++  addr := uintptr(unsafe.Pointer(addrObj))
7 ++  causeEscape((*BO)(unsafe.Pointer(addr)))
8 }
```

Listing 3: Optimization Procedure

3.4 Correctness Verification Phase

When the pointed object is released before it is still needed in another thread, this situation could be a bad optimization. Thus, we use verification methods to decide its correctness. The verification method adopts an effective and secure strategy to make decisions on verification. In this module, we present a static verification method to ensure the memory integrity of the optimized code. The validation of memory integrity is challenging from two major perspectives:

- *Sensitive to bad cases.* We do need to apply strict verification to ensure the correctness of optimized code.
- *Compatible with multi-threaded code.* Golang is widely used for multi-threaded execution. Verification need to be compatible with multi-threaded situations.

Details of the verification methods are shown in Algorithm. 2. The algorithm takes a Context object "ctx" as input, and its goal is to finish verification tasks. The algorithm adopts two strategies to verify the memory integrity: AST-based and llvm IR-based verification. AST (short for abstract syntax tree) is a simple and common data structure for program analysis. The AST-based method (Line.3-6) is to check whether the function call is a synchronous one. The pointers, which are passed to synchronous function call as arguments, will not be released before the call returns back. It is because the function call will not create a new thread. The optimization in this case is correct. Therefore, by scanning the abstract syntax tree, we can make decisions if the function call does not reach a multi-threaded condition. In Line.5, we do this check.

However, when the function call is an asynchronous one, things will change. AST-based verification can not make decisions. Then we use llvm IR-based verification (Line.8-21). First we parse the source file to llvm IR, which is an intermediate format of code. Then we scan IRs by using regular expressions in Line.9-14. Dealing with asynchronous cases, we adopted a simple but secure strategy: if the pointer variable is used in the asynchronous function call, no matter reading or writing, the optimization is determined to be a bad case (Line.17-18). Otherwise, the optimization is still considered as a good case (Line.19-20).

Algorithm 2 Memory Integrity Verification

Input:
Context to record run-time information, ctx
Output:
Save verification results in ctx

1: **for** each $filePath \in$ **do**
2: **for** each $ec \in ctx.EscapeCases[filePath]$ **do**
3: $ast \Leftarrow parseAST(filePath)$
4: $node \Leftarrow findFuncCall(ast, ec.LineNo)$
5: **if** $node$ is a synchronous call **then**
6: $ec.IsCorrect \Leftarrow TRUE$
7: **else**
8: $llvmIR \Leftarrow$ Run Gollvm to generate
9: $rF \Leftarrow$ a regular expression for file ID
10: $rV \Leftarrow$ a regular expression for variable ID
11: $rFun \Leftarrow$ a regular expression for function body
12: $fileID \Leftarrow rf.Find(filePath)$
13: $varID \Leftarrow rV.Find(fileID, ec.PtrName)$
14: $funcBodies \Leftarrow rFun.FindAll()$
15: $funcBodyMain \Leftarrow$ filter the function containing the escaped variable from $funcBodies$
16: $funcBodyGoroutine \Leftarrow$ filter the function using the escaped variable as argument from $funcBodies$
17: **if** $funcBodyGoroutine$ uses $varID$ **then**
18: $ec.IsCorrect \Leftarrow FALSE$
19: **else**
20: $ec.IsCorrect \Leftarrow TRUE$
21: **end if**
22: **end if**
23: **end for**
24: **end for**
25: **return** ctx

4 EVALUATION

For evaluation, we validate the proposed approach in two aspects.

- **How is the optimization effect of memory usage?** In the industrial area, projects are usually running under a certain memory size limit. A project's running process could be aborted when the memory usage exceeds the limit. Therefore, optimizing the total memory usage is significant in practice. Through comparing metrics of memory between the optimized and the original code, we can evaluate how is the optimization effect of memory usage.
- **What is the impact on speed of code execution?** Memory optimization effects are important. So is the running time consumption. To evaluate the performance of our approach on the speed of code execution, we make a comparison on the time consumption.

4.1 Experiment Setup

Data Sets. We choose real-world projects to construct our data sets. Some projects are collected from Github. We will present the optimization performance on these open-source programs. Message passing is a popular technique concurrency and object-oriented programming. We select a number of Golang projects in the field of message passing, to form the open-source data set. Table. 2 shows the detailed information. We choose 10 Golang projects. Column "LOC" means the lines of Golang code. Some projects contains dependent packages (a.k.a vendor). Notes that "LOC" only counts code outside the dependency packages. These projects have an average of 1,808 lines of code, excluding kubernetes.

NO	PROJECT	LOC
0	kubernetes/kubernetes	3,215,777
1	draftcode/sandal	8,741
2	conictus/wfe	2,489
3	armon/relay	2,387
4	gofort/dispatcher	1,370
5	crosbymichael/message	438
6	Zilog8/hgmessage	342
7	mediocregopher/ghost	248
8	FreekingDean/gotWrap	194
9	Shailjakant12/Message	67

Table 2: Open-Source Data Set.

Meanwhile, we also collect some industrial projects, which are used in companies. Our approach can help to optimize memory usage in industrial practice. According to the statistics of Bytedance Inc., there were 114 memory-related online accidents in 2018. Today, more and more Internet companies are concerned about memory management and memory optimization issues. To evaluate the optimization effect in the industrial data set, we apply our approach on industrial backend projects of an application in Bytedance Technology. Table. 3 shows information about our industrial data set. The size of projects varies from 287,364 to 293 lines of Golang code.

Criterion. To evaluate the performance of our memory optimization approach, we apply our approach to the data sets. For each Golang project, we calculate the metrics in two dimensions: memory usage, and time consumption. Detailed metrics are shown in Table. 4. The 1-7 metrics concerns on the memory, which are all in Bytes. The last two metrics are related to time consumption, which are in Nanoseconds.

NO	PROJECT	LOC	NO	PROJECT	LOC
1	goapi	287,364	9	noops	33,978
2	bookshelf_api	169,330	10	search	27,720
3	decorator	130,408	11	challenge	23,995
4	feed	87,036	12	pusher	17,826
5	trade	67,143	13	coreuser	9,040
6	account	64,838	14	user_achieve	8,729
7	content_v2	61,268	15	goods	7,967
8	content	37,626	16	admin_goapi	293

Table 3: Industrial Data Set.

Dimension	Metric	Description
Memory	1.Alloc	bytes of allocated heap objects
	2.TotalAlloc	cumulative bytes allocated
	3.Sys	bytes of memory got from OS
	4.Mallocs	cumulative count of heap object
	5.HeapAlloc	bytes of allocated heap object
	6.HeapInUse	bytes in in-use spans
	7.HeapObjects	number of allocated heap object
Time	8.PauseTotalNs	cumulative time of GC pause
	9.TC	time consumption of execution

Table 4: Metrics in Evaluation

Execution Platform. Most experiments are conducted on a MacBook Pro with a 2.5GHz Intel Core i7 processor and 16 GB memory, and the Golang version is 1.9.3 darwin/amd64. The experiment on Golang project "kubernetes" is conducted on a Ubuntu server with a 3.6GHz Intel Core i7 processor and 64 GB memory.

4.2 Results on Open-Source Data Set

How is the optimization effect of memory usage? Figure. 4 shows the optimized cases on open-source data set. In each table, column "Metrics" lists the nine metrics to evaluate the performance in memory optimization and time consumption. Column "Before" shows the cost of metrics for the original code. Column "After" shows the cost of metrics for the optimized code. It should be noted that most of these open-source projects are continuous execution. For example, in Figure. 4c, this code runs as an unstoppable loop to consume messages in a message broker like RabbitMQ. Therefore, Column "Before" and "After" are defined as the change of metrics after a specific function call, in which we do optimizations. These two columns of data represent the memory consumption and runtime of the function before and after the call.

The most important column in Figure. 4 is "Change". This column shows the comparison between "Before" and "After". It represents the effect of our optimization approach. For example, in Figure. 4e, Row "HeapObjects" shows the number of allocated heap object. The number changes from 6,737 to 4,096, reduced by 39.20%. It should be noted that we have bolded the data with an optimization ratio of

(a) Proj.1: lang/parsing/parser.go,l

Metrics	Before	After	Change
Alloc	101376	101312	0.06%↓
TotalAlloc	101376	101312	0.06%↓
Sys	1740800	1740800	-
Mallocs	195	194	0.51%↓
HeapAlloc	101376	101312	0.06%↓
HeapInUse	163840	163840	-
HeapObjects	163	162	0.61%↓
PauseTotalNs	0	0	-
TC	370176	360192	2.70%↓

(b) Proj.2: resultstore.go, buffer

Metrics	Before	After	Change
Alloc	26024	25912	0.43%↓
TotalAlloc	26024	25912	0.43%↓
Sys	3903488	3903488	-
Mallocs	266	265	0.38%↓
HeapAlloc	26024	25912	0.43%↓
HeapInUse	16384	16384	-
HeapObjects	233	232	0.43%↓
PauseTotalNs	0	0	-
TC	2370048	2229760	**5.92%**↓

(c) Proj.3: consumer.go, msg

Metrics	Before	After	Change
Alloc	6632	4584	**30.88%**↓
TotalAlloc	6632	4584	**30.88%**↓
Sys	3346432	3346432	-
Mallocs	66	62	**6.06%**↓
HeapAlloc	6632	4584	**30.88%**↓
HeapInUse	16384	16384	-
HeapObjects	50	46	**8.00%**↓
PauseTotalNs	0	0	-
TC	360192	280064	**22.25%**↓

(d) Proj.4: worker.go, task

Metrics	Before	After	Change
Alloc	6560	6048	**7.80%**↓
TotalAlloc	6560	6048	**7.80%**↓
Sys	3608576	3608576	-
Mallocs	104	86	**17.31%**↓
HeapAlloc	6560	6048	**7.80%**↓
HeapInUse	24576	24576	-
HeapObjects	79	63	**20.25%**↓
PauseTotalNs	0	0	-
TC	290048	189952	**34.51%**↓

(e) Proj.5: main.go, newFeed

Metrics	Before	After	Change
Alloc	370120	244696	**33.89%**↓
TotalAlloc	3.039E7	3.038E7	0.02%↓
Sys	6686968	6686968	-
Mallocs	740727	740641	0.01%↓
HeapAlloc	370120	244696	**33.89%**↓
HeapInUse	851968	729088	**14.42%**↓
HeapObjects	6737	4096	**39.20%**↓
PauseTotalNs	433692	304960	**29.68%**↓
TC	1.334E9	1.312E9	1.65%↓

(f) Proj.6: receiver.go, p

Metrics	Before	After	Change
Alloc	1.07E6	1.06E6	0.366% ↓
TotalAlloc	2.77E8	2.77E8	0.002% ↓
Sys	2.77E8	2.77E8	0.095% ↓
Mallocs	4911	4889	0.448% ↓
HeapAlloc	1.07E6	1.06E6	0.366% ↓
HeapInUse	1.23E6	1.18E6	4.000% ↓
HeapObjects	73	64	**12.329%** ↓
PauseTotalNs	2.54E6	2.00E6	**21.079%** ↓
TC	4.12E8	3.94E8	4.171% ↓

(g) Proj.7: listen.go, msgwrap

Metrics	Before	After	Change
Alloc	17960	17896	0.36%↓
TotalAlloc	17960	17896	0.36%↓
Sys	3442936	3442936	-
Mallocs	247	245	0.81%↓
HeapAlloc	17960	17896	0.36%↓
HeapInUse	40960	40960	-
HeapObjects	220	218	0.91%↓
PauseTotalNs	0	0	-
TC	1.005E9	1.004E9	0.14%↓

(h) Proj.8: tls_client.go, config

Metrics	Before	After	Change
Alloc	176936	176392	0.31%↓
TotalAlloc	176936	176392	0.31%↓
Sys	1081344	1081344	-
Mallocs	1662	1659	0.18%↓
HeapAlloc	176936	176392	0.31%↓
HeapInUse	270336	196608	**27.27%**↓
HeapObjects	1418	1414	0.28%↓
PauseTotalNs	0	0	-
TC	1.07E7	9.66E6	**9.72%**↓

(i) Proj.9:subscriber.go,actualmsg

Metrics	Before	After	Change
Alloc	28120	26472	**5.86%**↓
TotalAlloc	28120	26472	**5.86%**↓
Sys	1.08E6	1.08E6	-
Mallocs	387	322	**16.80%**↓
HeapAlloc	28120	26472	**5.86%**↓
HeapInUse	122880	81920	**33.33%**↓
HeapObjects	323	274	**15.17%**↓
PauseTotalNs	0	0	-
TC	8.75E6	8.38E6	4.23%↓

Figure 4: Optimized Effect in Open-Source Projects.

more than 5 percent. As shown in the tables, there are some cases, in which the memory optimization effects are much obvious, such as Figure. 4i, 4c, 4e, 4d. In other cases, the amount of memory used is large, so the base figure is larger. Therefore, the optimization ratios could hardly reach a high level. For example, in Figure. 4f, "TotalAlloc" exceeds 276 million bytes (about 263 MB), which is too much larger than other cases. In this case, even if we save 5,000 bytes, the optimization ratio is only 0.002%.

It is indeed necessary to evaluate larger benchmarks. We run our tool on Kubernetes [2]. Kubernetes has 3,215,777 lines of Golang code, which consists of 2,127 Golang packages. It takes 82 minutes and 4 seconds to finish the entire optimization procedure. We find 32 optimized cases [4] in total, proving that our method also works well with big code.

Through analyzing the memory optimization effects on open-source data set, we can conclude that our optimization methods can achieve optimization effects on the various metrics used in memory. Among the groups, the heap allocation is reduced by 0.06%-33.89%, with 8.88% average. The heap usage is reduced by 0%-33.33%, with 8.78% average.

What is the impact on the speed of code execution? On the issue of memory optimization, we should not only focus on the memory metrics, but also the impact on the speed of the code. As shown in Figure. 4, the last 2 rows in each table are related to RQ2. Row "PauseTotalNs" shows the cumulative time of GC pause. Garbage collection (GC) causes stop-the-world. Therefore, we try to reduce the frequency of GC. As shown in the table, "PauseTotalNs" is zero in some cases, such as Figure. 4i, 4h, etc. This is because the heap memory used in the execution of the code did not reach the expected upper limit, so the garbage collection mechanism in Golang was not triggered. Figure. 4e is an example where "PauseTotalNs" is reduced. In this case, the time of GC pause is saved by 29.68%. In addition, we focus on the time consumption of code execution. As the table shows, in the best set of experimental results, the code runs down by 34.51% in Figure. 4d.

[4]Detailed information of optimized cases: https://github.com/wangcong15/escape-from-escape-analysis-of-golang/blob/master/res/case-kubernetes.txt

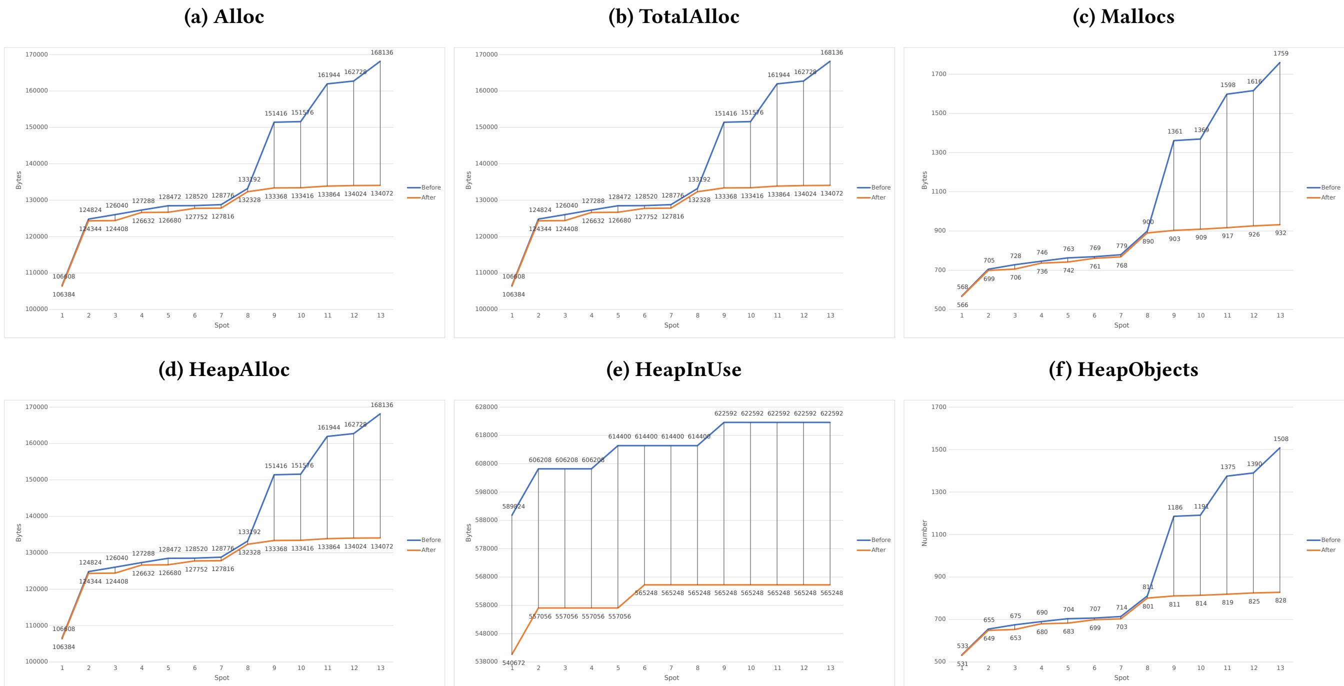

Figure 5: Results: Case Study I on Memory Optimization.

Through analyzing the execution speed effects on open-source data set, we can conclude that the proposed optimization method can achieve positive effects. Among the groups of optimization, time consumption is reduced by 0.14%-34.51%, with 9.48% average. The cumulative time of GC pause is reduced by 0%-29.68%, with 5.64% average.

As for the memory integrity verification, all are returned with true value. Furthermore, we also apply Go-Fuzz to study the execution of the optimized code and the original code, and their execution results are consistent with each other for 24 hours execution.

```
1  func ReceiveData() {
2      ... // spot.1 - spot.5
3      for { // here is a loop to handle messages
4          ... // spot.6 - spot.7
5          cw_pprof.PrintMemBrief() // spot.8
6          err = json.Unmarshal(m.Body, &actualmsg)
7          cw_pprof.PrintMemBrief() // spot.9
8          ... // spot.10 - spot.13
9      }
10 }
```

Listing 4: Code Snippet for Case Study I

Case Study. Listing. 4 shows an optimized case in open-source data set. The code comes from Proj.9 (Shailjakant12/Message). It is noteworthy that the variable "actualmsg" is moved to heap because its pointer is passed to function call in Line.6. For convenience, we call this original code as the group "Before". By applying our approach in this paper, we could avoid "actualmsg" being moved to the heap. We transform the code and call the optimized code as the group "After". In order to observe the memory changes in detail, we insert some spots (memory information printer) to track the status of memory. For each spot, we print the metrics of memory, including "Alloc", "TotalAlloc", etc.

Optimization effects of these metrics are shown in Figure. 5. The six figures illustrate the comparison between the original code and the optimized code. Let us take a look at the comparison in Figure. 5a, 5b, 5c, 5d, 5f. We try to feed a long string for "actualmsg". The large variable is moved to the heap, leading to the rapid rise in the 9th spot of the group "Before". On the other hand, the change of group "After" is smooth. With optimization, variable "actualmsg" should only stay in the stack.

In addition, Figure.5e shows the comparison on "HeapInUse". This metric counts the bytes in in-use spans. In fact, it is not synchronized with heap memory allocations. However, this metric is very significant. In the industrial area, projects are usually running under a certain memory size limit. We compare "HeapInUse" to decide which group brings more burden to system memory. Through analyzing this case in detail, we conclude that the optimization effect brought by our method is the same as expected.

4.3 Results on Industrial Data Set

Here, we will present the experimental results on the industrial data set. Figure. 7 shows the optimization results on industrial projects. For example, we find 123 optimized cases in the repository "goapi". By comparison, we find that there are relatively more optimized cases in the industrial data set. Through scanning the source code, we find that industrial projects have more complex package constructions and more frequent usage of objects and pointers. The above reasons lead to the larger size of optimized cases. All these cases are confirmed by Bytedance's senior developers.

(a) Alloc

Spot	Before	After	Change
1	0	0	-
2	2320	2000	13.79%↓
3	4400	4080	7.27%↓
4	128696	127384	1.02%↓

(b) TotalAlloc

Spot	Before	After	Change
1	0	0	-
2	2320	2000	13.79%↓
3	4400	4080	7.27%↓
4	128696	127384	1.02%↓

(c) Malloc

Spot	Before	After	Change
1	0	0	-
2	16	11	31.25%↓
3	29	24	17.24%↓
4	2253	2238	0.67%↓

(d) HeapAlloc

Spot	Before	After	Change
1	0	0	-
2	2320	2000	13.79%↓
3	4400	4080	7.27%↓
4	128696	127384	1.02%↓

(e) HeapInUse

Spot	Before	After	Change
1	0	0	-
2	0	0	-
3	0	0	-
4	49152	16384	66.67%↓

(f) HeapObjects

Spot	Before	After	Change
1	0	0	-
2	13	8	38.46%↓
3	22	17	22.73%↓
4	1839	1823	0.87%↓

Figure 6: Results: Case Study II on Memory Optimization.

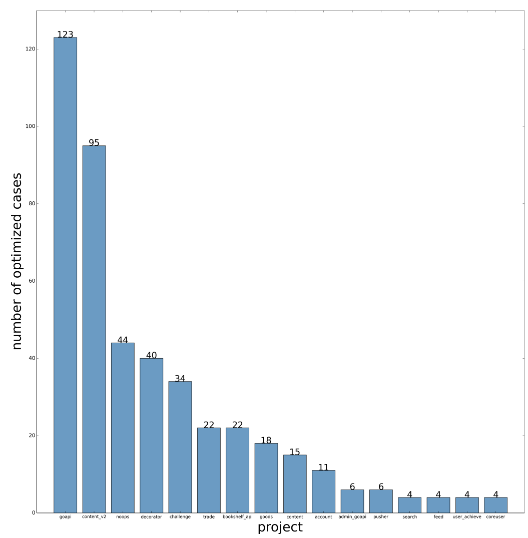

Figure 7: Number of Optimized Cases in Industrial Projects.

PROJ	FILE	LOC	VAR
account	dal/load.go	276	ugl
account	dal/load.go	295	ugl
account	dal/account.go	597	userContentR
account	dal/account.go	597	userCreateE
account	dal/agreement.go	78	updateE
account	dal/load.go	319	it
account	dal/load.go	358	rewards
account	rpc/rpc.go	43	req
account	rpc/rpc.go	64	req
account	rpc/rpc.go	64	num

Table 5: Opimized Cases in Industrial Code (Snaps).

Table. 5 shows a snapshot of optimized cases [5]. For example, in the 8th case, we optimize the variable "req". In backend code, "req" contains complex user information, including user city, app version, request parameters, etc. This variable could be a large one. However, in this case, the pointer of "req" is passed to a synchronous function call, making "req" move to heap memory. Our approach prevents "req" from moving to the heap, in order to reduce the heap allocation. These optimizations have been confirmed and adopted by Bytedance's senior developers.

[5]Due to the confidentiality requirements of enterprise data, we show only 10 examples of optimized Golang cases.

Case Study. Listing. 5 shows an optimized case in industrial data set. The code comes from Project.6 (Account). It is noteworthy that the variable "ugl" is moved to heap because its pointer is passed to function call in Line.7. For convenience, we call this original code as the group "Before". By applying our approach in this paper, we could avoid "ugl" being moved to the heap. We transform the code and call the optimized code as the group "After".

```
1  func LoadUserGoodsListFromDB(...) (...) {
2      cw_pprof.PrintMemBrief() // spot.1
3      ugl := []*models.UserGoods{}
4      cw_pprof.PrintMemBrief() // spot.2
5      queryOpt := db.QueryOption{/* ... */}
6      cw_pprof.PrintMemBrief() // spot.3
7      err := tradeDB.Load(ctx, &ugl, queryOpt)
8      cw_pprof.PrintMemBrief() // spot.4
9      return ugl, err
10 }
```

Listing 5: Code Snippet for Case Study II

Figure. 6 shows the results of this case. For the sake of comparison, we set the data of the first spot to zero. Column "Change" shows the comparison between "Before" and "After". It represents the effect of our optimization approach. We can take a look at data in the last spot. Variable "ugl" is moved to the heap, leading to the reduction in Spot.4. With optimization, the variable "ugl" should only stay in the stack. As shown in these six tables, metrics of memory usage are optimized through our approach.

4.4 Lessons Learned

From the study of optimization on escape analysis, we have learned three important lessons:

Optimization of Golang's escape analysis algorithm is a significant issue. As written in Golang's official documents, there are many compromises in Golang's escape analysis algorithm. Actually there could be a number of variables stored in heap which leads to a waste of memory. If large variables are moved to heap, the garbage collection will work frequently. This is very computationally intensive. Moreover, attackers even can make use of this weakness to explode the heap memory. Therefore, the issue is significant in practice. In our experiment, we find 452 optimized cases in industrial projects, which means that the waste of memory issue exists commonly, especially in large-scale Golang projects.

Guaranteed memory integrity is a very high priority requirement for Golang's memory optimization. Golang is a good and popular programming language, because of its well-designed concurrent programming grammar and fast execution speed. Memory management becomes a more complex task in concurrent programs compared to single-threaded ones. Thus, guaranteed memory integrity becomes more challenging in this situation. In order to ensure the correctness of optimization, we validate the optimized code by a verification method. In the procedure of verification, we analyze the control flow graph and data flow of programs, and decide whether the optimization does harm to original programs. The optimized code should not go wrong, or even change the behavior. The verification method proves statically and theoretically that the escaped variable will not be released before its read and write operations end. In this way, the verification method decides that execution of optimized code will not crash or change. In our experiments, there is not a optimized cases failing for memory integrity issues. Most programmers would rather waste the memory than introduce risks into programs. That's why we work a lot in memory integrity validation. We optimize the code and prove that the optimized code is correct.

Changing the code is an effective way for memory optimization. We are seeking an alternative approach to reduce redundant memory waste. Optimization is helpful. Changing program statements can achieve the goal of optimization. In this paper, we propose an approach to optimize Golang's escape analysis. E.g., we use "uintptr" to transform ordinary pointers into "unsafe.Pointer", in order to bypass Golang's escape analysis. Experimental results demonstrate that we are able to successfully "escape" from escape analysis of Golang, and save the resource consumption with optimization.

5 CONCLUSION

In this paper, we have proposed a memory optimization algorithm on escape analysis for Golang. We have presented a framework that integrates escape variable positioning, memory optimization, and memory integrity insurance. The implemented prototype works well in both open-source and industrial projects. The experimental results demonstrate the correctness and effectiveness. It is able to successfully analyze and transform Golang programs to optimize the original escape analysis mechanism, and eventually reduce the heap memory usage and speed up code execution.

In the future, we will continue to study the memory optimization problem in two directions: to support more escape algorithm recognition patterns, and to support more programming languages and projects. For the first direction, we have introduced the pattern in which a pointer variable is used as a function argument. We will explore in depth for other patterns. For the second direction, we are currently optimizing memory for Golang's escape algorithm and try to customize the approach to other programming languages. Furthermore, we will also try to apply the approach to more open source projects to improve the robustness of the work.

REFERENCES

[1] William A Arbaugh, Nick Louis Petroni Jr, Timothy Jon Fraser, and Jesus Maria Molina-Terriza. 2015. Method and system for monitoring system memory integrity. (Feb. 10 2015). US Patent 8,955,104.

[2] David Bernstein. 2014. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing* 1, 3 (2014), 81–84.

[3] Bruno Blanchet. 1998. Escape analysis: Correctness proof, implementation and experimental results. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 25–37.

[4] Bruno Blanchet. 1999. Escape analysis for object-oriented languages: application to Java. In *Acm Sigplan Notices*, Vol. 34. ACM, 20–34.

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

[6] Chien-Wen Chen and Che-Yueh Kuo. 2019. Memory management method, memory control circuit unit and memory storage device. (Jan. 10 2019). US Patent App. 15/690,286.

[7] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. *Acm Sigplan Notices* 34, 10 (1999), 1–19.

[8] Jong-Deok Choi, Manish Gupta, Mauricio J Serrano, Vugranam C Sreedhar, and Samuel P Midkiff. 2003. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25, 6 (2003), 876–910.

[9] Alan Deutsch. 1989. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 157–168.

[10] Alain Deutsch. 1997. On the complexity of escape analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 358–371.

[11] Wei Feng, Xiaohua Shi, and Wenru Wang. 2018. Eliminating object reference checks by escape analysis on real-time Java virtual machine. *Cluster Computing* (2018), 1–12.

[12] Golang. cited 2019. Golang: Compiler Optimizations. https://github.com/golang/go/wiki/CompilerOptimizations. (cited 2019).

[13] Benjamin Goldberg and Young Gil Park. 1990. Higher order escape analysis: optimizing stack allocation in functional program implementations. In *European Symposium on Programming*. Springer, 152–160.

[14] John Hannan. 1998. A type-based escape analysis for functional languages. *Journal of functional programming* 8, 3 (1998), 239–273.

[15] Patricia M Hill and Fausto Spoto. 2002. A foundation of escape analysis. In *International Conference on Algebraic Methodology and Software Technology*. Springer, 380–395.

[16] Jeff Huang. 2016. Scalable thread sharing analysis. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 1097–1108.

[17] Richard Jones and Rafael Lins. 1996. *Garbage collection: algorithms for automatic dynamic memory management*. Vol. 208. Wiley Chichester.

[18] Thomas Kotzmann and Hanspeter Mössenböck. 2005. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. ACM, 111–120.

[19] Young Gil Park and Benjamin Goldberg. 1992. Escape analysis on lists. In *ACM SIGPLAN Notices*, Vol. 27. ACM, 116–127.

[20] Alexandru Salcianu and Martin Rinard. 2001. Pointer and escape analysis for multithreaded programs. *ACM SIGPLAN Notices* 36, 7 (2001), 12–23.

[21] Yulei Sui, Peng Di, and Jingling Xue. 2016. Sparse flow-sensitive pointer analysis for multithreaded programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 160–170.

[22] Yan Mei Tang and Pierre Jouvelot. 1992. Control-Flow Effects for Escape Analysis.. In *WSA*. 313–321.

[23] Frédéric Vivien and Martin Rinard. 2001. Incrementalized pointer and escape analysis. In *PLDI*. Citeseer, 35–46.

[24] John Whaley and Martin Rinard. 1999. Compositional pointer and escape analysis for Java programs. *ACM Sigplan Notices* 34, 10 (1999), 187–206.

[25] Paul R Wilson. 1992. Uniprocessor garbage collection techniques. In *Memory Management*. Springer, 1–42.