Pablo Martins Musa

Profiling Memory in Lua

Dissertação de Mestrado

Dissertation presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática, PUC-Rio as partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Roberto Ierusalimschy

Rio de Janeiro
June 2015

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Pablo Martins Musa

Profiling Memory in Lua

Dissertation presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática do Centro Técnico Científico da PUC-Rio, as partial fulfillment of the requirements for the degree of Mestre.

Rio de Janeiro, June 19th, 2015

Pablo Martins Musa

Pablo Martins Musa graduated from PUC-Rio in Computer Engineering. He is currently working at Elastic as an Educational Engineer where he works on search and analytical engines for distributed systems.

Bibliographic data

Musa, Pablo Martins

Profiling Memory in Lua / Pablo Martins Musa; advisor: Roberto Ierusalimschy . — 2015.

89 f. : il. ; 30 cm

1. Dissertação (Mestrado em Informática) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2015.

Inclui bibliografia

1. Informática – Teses. 2. Lua. 3. Perfilador de Memória. 4. Inchaço de Memória. Linguages de Script. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

First of all I would like to thank my family for their continuous support. I would never have gotten so far if it wasn't for them and for the education they gave me.

I am very grateful to my friends André Maidl and Alexandre Skyrme for the endless ours at LabLua, but specially for all the text reviews that made this work possible.

I am very grateful for the funding from CAPES, without which this work would not have been possible. I also would like to thank Google Summer of Code for partially funding this work. Finally, I am very much indebted to PUC-Rio whose generous scholarship made my graduation possible.

Abstract

Musa, Pablo Martins; Ierusalimschy, Roberto (Advisor). **Profiling Memory in Lua**. Rio de Janeiro, 2015. 89p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Memory bloat is a software problem that happens when the memory consumption of a program exceeds the programmer's expectations. In many cases, memory bloat hurts performance or even crashes applications. Detecting and fixing memory bloat problems is a difficult task for programmers and, thus, they usually need tools to identify and fix these problems. The past two decades produced an extensive research and many tools to help programmers tackle memory bloat, including memory profilers. Although memory profilers have been largely studied in the last years, there is a gap regarding scripting languages. In this thesis, we study memory profilers in scripting languages. First, we propose a classification in which we divide memory profilers in manual and automatic, based on how the programmer uses the memory profiler. Then, after reviewing memory profilers available in three different scripting languages, we experiment some of the studied techniques by implementing two automatic memory profilers to help Lua programmers deal with memory bloat. Finally, we evaluate our tools regarding how easy it is to incorporate them to a program, how useful their reports are to understand an unknown program and track memory bloats, and how much overhead they impose.

Keywords

Lua; Memory Profiler; Memory Bloat; Scripting Languages.

Resumo

Musa, Pablo Martins; Ierusalimschy, Roberto. **Analizando o uso de Memória em Lua**. Rio de Janeiro, 2015. 89p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Inchaço de memória é um problema que ocorre quando a memória consumida por um programa excede a expectativa do programador. Em muitos casos, o inchaço de memória prejudica o desempenho ou, até mesmo, interrompe a execução de aplicações. Detectar e consertar inchaços de memória é uma tarefa difícil para programadores e, portanto, eles costumam usar ferramentas para identificar e consertar problemas desta natureza. Nas últimas duas décadas, muitos trabalhos e ferramentas foram desenvolvidos com o intuito de ajudar programadores a abordar problemas de inchaço de memória, entre eles perfiladores de memória. Apesar de perfiladores de memória terem sido muito estudados nos últimos anos, existe uma lacuna em relação a linguagens de script. Nessa dissertação, nós estudamos perfiladores de memória para linguagens de script. Primeiro, nós propomos uma classificação que divide as ferramentas em manual e automática baseada em como elas são usadas pelos programadores. Em seguida, após estudar ferramentas disponíveis em três linguagens de script diferentes, nós experimentamos algumas das técnicas estudadas ao construir dois perfiladores de memória automáticos para ajudar programadores Lua a resolver inchaços de memória. Finalmente, nós avaliamos ambas as ferramentas com relação a facilidade de integração ao programa, a utilidade dos relatórios para o entendimento de programas desconhecidos e para a localização de inchaços de memória e ao custo de desempenho que elas geram.

Palavras–chave

Lua; Perfilador de Memória; Inchaço de Memória; Linguages de Script.

Contents

List of Figures

How much truth does a spirit `endure`*, how much truth can it* `dare`*? This became for me more and more the actual test of value.*

Friedrich Nietzsche

1 Introduction

Memory bloat is a software problem that happens when the memory consumption of a program exceeds the programmer's expectations. Memory bloat can happen due to multiple factors, such as excessive object[1] creation, data structure misuse, and not deleting references that point to objects that will not be used anymore [32]. In many cases, memory bloat impacts performance or even crashes applications. To create and garbage-collect excessive objects increases the program execution time. In long running programs, the growing number of objects may cause crashes due to out of memory errors. Major issues regarding memory bloat have been reported in popular applications, such as Firefox [43], Internet Explorer [24], Tomcat [41], and Gmail [23].

Detecting and fixing memory bloat is a difficult task [16]. For instance, out of memory errors may happen in parts of a program that have no direct relation to the cause of the bloat. Searching for the cause of a memory bloat problem is similar to searching for the cause of a performance bottleneck. In both cases, it is hard to pinpoint where the problem is located at, and even experienced programmers usually get it wrong [8]. Often, just by analyzing the source code, it is very hard to spot memory bloat. Therefore, programmers need tools to help them identify and fix memory bloat problems.

In the past three decades *developers* produced many tools to help *programmers* tackle memory bloat [33, 11, 20, 17, 3, 18, 30, 4].[2] To tackle memory bloat, programmers can use *general* or *specific* tools. General tools give an overview of the memory behavior of a program and help programmers look for hot spots. As an example, `mprof` [33] summarizes the allocated memory by function. When using general tools, problems are usually solved by a repetitive process of profiling and testing. Specific tools address particular problems in a more automated manner and produce detailed information in a fine-grained level. `Cork` [14], for example, detects systematic heap growths based on the

[1]Throughout this thesis we use *object* loosely, to include any kind of structured data record, such as Pascal records or C structs, as well as full-fledge objects with encapsulation and inheritance, in the sense of object-oriented programming.

[2]Throughout this thesis we use *developer* to refer to the creator of a tool and *programmer* to refer to the user of a tool.

information collected from multiple garbage-collections. On the one hand, `Cork` does not show general information, such as the total memory used by the program or the number of allocated objects. On the other hand, `Cork` identifies that the program has a potentially harmful heap growth and shows the objects that are likely causing it. Unfortunately, specific tools produce many false positives and usually demand virtual-machine modifications. Looking back over more than ten years of research shows that the industry has been rejecting specific tools in favor of general tools [28]. Among general tools, *memory profilers* play an important role.

Memory profilers, as the name implies, are profilers that focus on the memory. Profilers are popular tools that help programmers better understand the behavior of a program. Profilers are important tools for optimization and locating bad resource utilization [8]. Accordingly, memory profilers help programmers understand different memory aspects of a program. Memory profilers collect memory information and output it in a insightful format. We once more use `mprof` as an example: it monitors allocated and deallocated blocks during a program execution and outputs a list of blocks that were not deallocated before the end of the program. Another example is `Kbdb` [26], which allows the programmer to pause execution and inspect the heap through a graphical display.

Although memory profilers have been largely studied in the last years, there is a gap regarding scripting languages. Scripting languages, such as Python, JavaScript, and Lua, have been largely adopted in the last years [44] and, along with them, programmers are facing many memory bloat problems [39, 49, 36]. Although there are many online discussions about identifying and fixing memory bloats in these languages (e.g. blog posts, mailing lists, forum questions), we could find few memory profilers for these languages and only one article [23] that discusses the topic.

In this thesis we study memory profilers in scripting languages. First, we propose a classification in which we divide memory profilers into *manual* and *automatic* based on how the programmer uses the memory profiler. We define manual profilers as tools that collect data at manually defined lines of a program. Manual profilers are usually simple tools used to inspect objects and to show information in a pretty format. We define automatic profilers as tools that collect data automatically during a program execution. Automatic profilers are usually more complex than manual profilers and can be used to understand the overall memory behavior of a program.

After the proposed classification, we review published work on memory profilers and analyze memory profilers available in three different scripting languages: JavaScript, Python, and Lua. We chose these languages because they are popular, different from each other, and, as we said before, have many reports about memory bloat. After reviewing memory profilers, we experiment some of the studied techniques by implementing two automatic memory profilers to help Lua programmers deal with memory bloat.

The first memory profiler we present is `luamemprofiler`. It was developed to explore three techniques: real-time[3] visualization, type/class data categorization, and ongoing interaction. Real-time visualization, also used by Detlefs and Kalsow [7], gives different views of the program allocation, such as how much storage of each type is being allocated since the beginning of the program execution or since the last screen update. Type/class data categorization, also used by Sun and Gehringer [29], helps the programmer identify structures that are responsible for most of the allocated memory. Finally, ongoing interaction with the program, also used by Serrano and Boehm [26], provides the opportunity to pause the program to analyze information such as live objects and the stack trace, among others.

The second memory profiler we present is `lmprof`. It was created to explore `gprof`'s [10] largely used model which summarizes events based on function calls. `lmprof` focuses on memory allocation: for each memory block allocated during the program execution, `lmprof` records the size, the current executing function, and the specific function that called the current function (hereafter called *parent-function*). At the end of the profiled program, `lmprof` saves the recorded information into a file. Then, a separate process analyzes the file and consolidates a report that, among other informations, lists the total memory allocated by each function.

The remainder of this thesis is organized as follows. Chapter 2 presents a profiler classification and examines existing memory profilers. Chapter 3 discusses the guidelines used to develop both `luamemprofiler` and `lmprof` and, then, details each tool separately. Chapter 4 evaluates both tools regarding ease of use (how easy it is to incorporate a profiler into an existing application), insights (how valuable the reports are to understand a program and track memory bloats), and performance. Finally, in Chapter 5, we conclude this thesis and discuss future work.

[3]Throughout this thesis we use *real-time* to refer to software that shows information almost immediately.

2
Related Work

In this chapter, we discuss relevant memory profilers to our work. We begin by presenting a classification for the discussed memory profilers. Then, we review memory profiler tools in four different sections. First, we examine published work on memory profiler techniques. Then, we explain JavaScript memory profilers that are currently used. After that, we present an overview of existing Python tools. Finally, we analyze memory profilers for Lua.

2.1
Classification

Profilers (not only for memory) have been the subject of extensive research since the early 70's [15]. Ever since, researchers have proposed different profiler types, such as statistical [10], overall [1], and input-sensitive [6].

In this thesis, we classify memory profilers based on the method used by the programmer to profile memory information of the target program. We divide memory profilers in two groups, *manual* and *automatic*. Our manual group is similar to the manual type described on Wikipedia [42]. However, our classification is simpler and focuses on the programmer, instead of on the developer. In the next sections we detail manual and automatic profilers.

2.1.1
Manual Profilers

Manual profilers are tools that collect data at manually defined lines of the target program. Manual profilers are usually used in a continuous program inspection process, similar to the debug process. Manual profilers implement a small set of functions that help programmers manually inspect memory at any point during the program execution. For example, the programmer can inspect how many objects are alive, what is the relationship among them, and how much memory is being used. Manual profilers are motivated by the perception that many programmers write their own pretty-printers or data dumpers to help them understand their programs data, but these code fragments are error-prone and cannot handle complex situations and types.

Imagine a programmer that wants to understand why a hash map is using too much memory. At some point, he creates a small chunk of code to print the hash map keys and values. The code may not print the complete information of values that are also hash maps. If it does, it will probably not handle cycles (which may cause an infinite loop) or it might not print the information in a format that is easy to read. Manual profilers try to avoid these kinds of problems by offering complete and tested functions that can be easily used to inspect objects and to show information in a pretty format.

2.1.2 Automatic Profilers

Automatic profilers are tools that collect data automatically during the target-program execution. Automatic profilers are used to analyze the overall behavior of a program. In contrast to manual profilers, which are used to inspect the program at an exact moment, automatic profilers are used to understand the behavior of a program during an execution interval. As an example, the programmer can change the allocation function to monitor how much memory is allocated by each function during the target-program execution. In automatic profilers, profiling happens in two sequential phases: *track* and *report*. The first phase, done during program execution, tracks program execution and collects data (hereafter called *metadata*). The second phase, done during or after program execution, consolidates the metadata and generates reports. We next detail track and report phase techniques that are relevant to our work.

Track Techniques

Tracking is the process of monitoring the execution of a program and collecting specific data. To track a program, automatic memory profilers usually instrument (add instructions to) the target program to collect the required information. *Hooking* is a very common instrumentation technique used to intercept specified function calls, messages, and events. Regarding automatic memory profilers, the main tracking method is to hook functions. There are two major function hooking techniques.

One technique, here named *call-hook*, consists in inserting hooks in the call protocol of the target-program functions. In some cases, another hook may be executed in the return sequence. The *call-hook* technique is useful to collect memory information at the function level, such as count the number of times that each function was called. These hooks can be inserted by different methods. For instance, to use `Python Memory Profiler` [21], the programmer annotates the target-program functions that will be hooked, so the compiler can properly insert the hooks.

The other technique, here named *allocation-hook*, consists in changing the default allocation functions of the target program (e.g. `malloc`, `free`, and `realloc`) into new functions that perform the allocation or deallocation and gather metadata. The *allocation-hook* technique is useful to collect memory information at the memory block level, such as which object-class is being allocated. There are different methods for changing the allocation function. We once more use `mprof` as an example: to use it, the programmer recompiles the target program with a special library that implements modified versions of `malloc` and `free`.

Report Types

Report is the output generated by the memory profiler after it consolidates the metadata. There are two main report types for an automatic memory profiler: *continuous-report* and *final-report.*

Continuous-report consists in reporting memory information during the target-program execution. In continuous-report, the output is continuously updated so the programmer can check memory operations as they happen. Continuous-report has a high performance and memory usage impact and provides information during the program execution. The latter allows the programmer to understand the memory flow of the target program.

Final-report consists in reporting memory information after the target-program execution. In final-reports, the output can be a summary or a trace of memory events. Summary is the most common one. It has low impact on performance and memory usage and provides an overview of memory usage, which allows the programmer to spot glaring errors. Trace has a high impact on performance and memory usage and provides all events that happened during execution. The latter allows the programmer to analyze specific parts of the target program.

Regardless of the report type, many memory profilers output contains, among other informations, the *shallow* and the *retained* size of the measured "property". Consider we are measuring memory allocated by functions. In this case, the memory profiler tracks the memory usage of all the functions in the target program. The shallow memory means how much memory a function allocates alone. The retained memory means how much memory a function and its descendants allocate. As an example, imagine a function `f1` that allocates 10 bytes and calls `f2`. Then, `f2` allocates 70 bytes and calls `f3`, which allocates 30 bytes. Finally, `f1` allocates more 40 bytes. The shallow sizes of `f1`, `f2`, and `f3` are 50, 30, and 70, respectively. The retained sizes of `f1`, `f2`, and `f3` are 150, 100, and 30 respectively.

2.2 Published Work

In this section, we discuss, in chronological order of publication, memory profiler tools that are relevant to our work. For each profiler tool, we summarize its functionalities, discuss how it is connected to the target program, explain relevant implementation details, and, finally, consider its overhead.

In 1982, Graham et al. [10] presented `gprof`, a CPU profiler for C, Fortran77, and Pascal. Although we are interested in *memory* profilers and `gprof` is a CPU profiler, many tools that we will discuss are based on `gprof`. Therefore, we first discuss `gprof` and, then, we discuss the other tools.

`gprof` is an automatic, call-hook, and final-report profiler. It is connected to the target program by recompiling the target program with a special compiler. The especial compiler hooks every function of the target program to gather metadata. The especial compiler also hooks the exit function to write the metadata into a file as the program exits.

As an automatic profiler, `gprof` has both track and report phases. For each function call during the track phase, the tracker gets the parent-function reference, forms a current-function/parent-function pair, and increments both the call counter and the execution time of the pair. The metadata is kept in a two-level hash table with the call site as the primary key and the current-function address being the secondary key. Saving the call counts is straightforward; the developer just needs to add or increment the metadata when the hook is executed. However, estimating execution time in time-sharing systems is difficult. `gprof` adopts a statistical approach, which we do not detail here because execution-time profilers are not the focus of this thesis. As the program finishes execution, the exit hook writes the metadata into a file that will be processed by the report phase.

The report phase is implemented as a separate program that processes the metadata to combine information on the tracked functions. While processing, `gprof` builds a dynamic call graph with arcs from callers to callees, and propagates times from descendants to ancestors by topologically sorting the graph. However, if the execution contains recursive calls, the call graph has cycles that cannot be topologically sorted. Cycles obscure allocation data among functions that compose the cycles and, thus, can be a problem to profilers that track function calls. In Chapter 3.3.1 we discuss cycles in more details. After processing the metadata and building the graph, `gprof` can produce two different profiles: *flat* and *call-graph*.

The flat profile consists of a list of all the functions that were called during the target-program execution, including the number of times they were called and for how many seconds they were executed. The latter information is divided in shallow and retained execution time. The functions are listed in decreasing order of the shallow execution time. The flat profile gives a quick overview of the called functions, and shows the functions that are themselves responsible for large fractions of the execution time.

The call-graph profile has almost the same entries as the flat profile, but for each function the call-graph profile details its direct parent- and child-functions. Another difference between the profiles is that the call-graph profile is sorted by the retained execution time. The call-graph profile can be seen as a textual graph and, thus, is useful to understand the connections (function calls) among the program functions.

Finally, the authors state that the tracking phase causes an execution time slowdown between 0.05x and 0.3x. They do not discuss the memory overhead.

In 1988, Zorn and Hilfinger [33] presented `mprof`, a profiler for C and Lisp that was inspired by `gprof` and extends its ideas to show the dynamic memory allocation data instead of execution times. Similarly to `gprof`, `mprof` tracks the memory operations of each function during program execution, writes the metadata into a file as the program exits, and processes this file in a separate program to produce different profiles.

`mprof` is an automatic, allocation-hook, and final-report memory profiler. `mprof` is connected to the target program by recompiling the target program with a special compiler. However, while `gprof` hooks every function call, `mprof` hooks only the allocation functions (`malloc` and `free` in C and `alloc_object` in Lisp). `mprof` also hooks the `exit` function to write the metadata into a file when the target-program exits.

During the track phase, in contrast to `gprof` that loads just the parent-function information and propagates this information to the ancestor functions in post processing, `mprof` traverses the entire function call chain to record the ancestor metadata. Because traversing the entire call chain is an expensive process, instead of recording the entire chain of callers, `mprof` amortizes the cost by breaking the call chain into a set of caller/callee pairs and by associating the bytes allocated with each pair in the chain. The authors state that there are a limited number of such pairs, even in very large C programs, so the `mprof` metadata is usually not big.

As `gprof`, `mprof`'s report phase reads the output file, consolidates the metadata to create a dynamic call graph, and prints a profile. There are two main differences between both tools. One difference is that `mprof` has four profile options instead of two. The "memory leak profile" presents a list of partial call paths that resulted in memory allocated but not subsequently freed. The "allocation bin profile" provides information about the sizes of objects that were allocated and which object-types correspond to each of the sizes. The "direct allocation profile" and the "allocation call-graph" correspond to the flat and the call-graph profiles generated by `gprof` with some modifications to handle memory information. The other difference is that, as `mprof` records the entire call chain, there is no need to propagate data from child to parent functions. However, cycles are also a problem. `mprof` handles cycles in the same way as `gprof` does, which we will detail in Chapter 3.3.1.

Zorn and Hilfinger [33] test `mprof` in four different programs and present the following result. The track phase presents a slowdown from 1.5x to 10x, a memory overhead of 33% maximum, and an output file that is less than 30KB. The average slowdown is between 2x and 4x, but can be very high in programs with long call chains.

In 1995, Detlefs and Kalsow [7] presented four tools used in SRC Modula-3 that aid in detecting and isolating storage management problems. They describe real memory problems they faced and detail all the tools that they created to deal with these problems, explaining how each tool was used to find or solve each problem.

The first tool is `Shownew`, which is an automatic, allocation-hook, and continuous-report memory profiler. `Shownew` shows a bar graph indicating how much storage of each type is being allocated along the program execution. The display is updated every `X` seconds, where `X` is parametrized. The programmer can also opt to see only the objects that were allocated since the last display update. `Shownew` is integrated into the runtime system, so that any SRC Modula-3 program can receive a special command-line argument that will cause it to run under the control of a `Shownew` process. In this case, there is no need for recompiling the target program as `Shownew` will dynamically change the allocation function into a function that counts the objects of each type and periodically forwards that information to the `Shownew` process.

The second tool is RTutils.Heap (`RTH`), which is a manual tool that reports the composition of the heap by data type. The programmer can use this tool to check the size of the heap and what kinds of objects are stored (how many objects and the total size of each type). To track programs with `RTH`, the programmer must modify the target program to periodically perform a garbage collection and then call `RTH`. `RTH` enumerates all the objects in the heap and classifies them by type, reporting the top 10 types ranked by the number of bytes they occupy. `RTH` also allows reports to be ordered by the number of allocated objects, and to limit reports to the top `N` types by the requested ranking method. Besides, the programmer can opt for a fine- or coarse-grained report. The former allows the programmer to select a type to breakdown by call site. The latter allows the programmer to select a type hierarchy report to see different subtypes accounting for different fractions of that total.

The third tool is RTHeapStats.ReportReachable (`RTR`), a manual tool that that tells how many bytes of storage are reachable from the *root set*, breaking down the roots in various ways. `RTR` executes “mini garbage-collections” to trace all reachable objects. Then, it ranks the modules by bytes reachable from their global variables and ranks the individual global variables by the amount of storage they reach. `RTR` is particularly useful to find out the root variable that is holding the memory bloat. To track programs with `RTR`, the programmer must modify the target program to periodically call `RTR`.

Finally, the fourth tool is RTHeapDebug (`RTD`), which is a manual tool that allows the programmer to verify the path of an object that should be collected, but is not. After suspecting of an object, the programmer can use `RTD` to print all paths from the root to the suspicious object.

Although the authors say that the performance of `RTR` seems quite acceptable according to their reports, they do not show these reports. Also, the authors do not show or comment about the performance of the other tools.

In 1997, Sun and Gehringer [29] developed an automatic, allocation-hook, and continuous-report memory profiler for IBM Smalltalk (hereafter called `smallmp`). `smallmp` monitors memory allocations and reports them to the user through a graphical display.

To capture all allocations in the entire Smalltalk image, `smallmp` extends the main allocation class and overwrites the four basic allocations methods. Every memory allocation is an event that is processed through a dynamic pipeline. The pipeline uses a producer/consumer model to easily compose a chain of event analyzers. Each event will go through the pipeline, get analyzed by the consumers and eventually dropped by the last consumer. For instance, an analyzer handling an allocation can filter the event, update the graphical display with the new event data, or record the event for further processing. The ability to filter events allows the programmer to discard events from specific classes, such as the allocations generated by the `smallmp` itself.

During program execution, there are two views for the programmer to understand the application's memory consumption behavior: the allocation matrix and the Memory-Allocation Graph Explorer (MAGExplorer). The allocation matrix provides a visual indication in the form of a matrix of how much memory each class used and which classes were responsible for allocating them. This visualization can be used to get a quick understanding of which classes consume the most memory in an application, which are the most frequently allocated objects, and which are the classes directly responsible for allocating them. The Memory-Allocation Graph Explorer allows the programmer to walk through the call graph viewing which paths are responsible for most of the allocated memory. Each class is represented as a node on the graph and colors indicate small or large memory allocations. MAGExplorer also allows users to interact with the graph by selecting objects to be detailed or querying a specific class name.

We would like to highlight three novelties of the work of Sun and Gehringer. First, instead of just numbers, they use colors to express how much memory was allocated by each class, which gives a good and fast overview of the program execution. Second, they allow the user to interact and query the call graph. Finally, the pipeline model allows different analysis to be performed with small code changes.

The authors present two simple tests. There is a slowdown of 2x just for loading the profiler, a slowdown from 3x to 28x for profiling specific classes and a slowdown from 20x to 180x for profiling all allocations.

In 2000, Serrano and Boehm [26] presented two tools for examining memory allocation in Scheme programs: `Kprof` and `Kbdb`. `Kprof` is an automatic memory profiler embedded in the regular Scheme time profiler, and was designed as a layer surrounding the `gprof` tool. `Kbdb` is a manual memory profiler for heap inspection that allows debugging application and real-time call-graph display.

`Kprof` is an automatic, call- and allocation-hook, and final-report memory profiler. One can think of `Kprof` as a `gprof` extension, with a nice front-end, for garbage collected languages. In addition to `gprof` metadata, each time a garbage collection is triggered, `Kprof` records the heap size, the number of live objects, and the number of allocations since the previous collection. The `Kprof` report includes an execution profile with common information, a trace of the heap size, and the time spent executing scheme functions, the garbage collection, and other C functions. Just like `gprof`, the programmer must recompile the target program to profile it with `Kprof`. Moreover, `Kprof` inherits some of `gprof` inaccuracies, such as statistical information and the lack of precise data in case the target program is optimized.

`Kbdb` is a heap inspection tool. At first, it acts as a debugger, so programs are run interactively and can be stopped to inspect variables. However, in addition to the debugger, the heap can be drawn in a graphical display where each pixel represents one cell of the heap. `Kbdb` displays the live objects and the chains of pointers that link these objects in the heap. Individual cells can be inspected by simply pointing at their corresponding pixel in the image. When a cell is inspected, `Kbdb` displays the Scheme type of its value, its allocation site, and its approximate age. Besides, based on de Pauw and Sevitsky [20], `Kbdb` uses the reference pattern technique, which makes repetitive sequences more understandable by eliminating redundancy and exposing their inherent structure. In other words, `Kbdb` groups similar objects (e.g. objects of the same class) to simplify the displayed information. `Kbdb` is embedded into a Scheme debugger also created by the Serrano and Boehm. The authors also modified the Boehm-Demers-Weiser garbage collector to provide back-pointer information, as part of the debug information that could already be associated with individual objects.

The authors present three tests comparing optimized code, profiled code (`Kprof`), and debugged code (`Kbdb`). While `Kprof` imposes a slowdown from 1.5x to 4.2x and no memory overhead, `Kbdb` imposes a slowdown from 2.9x to 9x and uses from 3.6x to 4.9x more memory.

2.3 JavaScript Tools

To detect and fix memory bloat in JavaScript programs, programmers usually use developer packages offered by web browsers. The majority of web browsers have built-in tools that use the browser JavaScript virtual-machine profiler and debug API to collect program information. Each web browser has its own virtual-machine implementation and its own profiler and debug API.

Chrome DevTools [48] seems to be the most used and complete developer package. Chrome DevTools is a set of web authoring and debugging tools built into Google Chrome. In this thesis, we focus on `Heap Profiler`, a tool that includes two different memory profilers: `Snapshot` and `Timeline`. `Heap Profiler` uses the Google Chrome graphical display and has one common window for both profilers. This common window is used to collect and analyze memory information. Next, we explain each memory profiler.

`Snapshot` is a manual memory profiler that allows the user to save the target-program heap information at any time. During the target-program execution, the programmer can take a snapshot of the heap. Each snapshot saves information about all the JavaScript objects that are currently alive and the relationships between these objects. The programmer can take as many snapshots as she wants.

`Timeline` is an automatic, final-report memory profiler. At any time during the target-program execution, the programmer can start recording memory information. When recording is activated, `Timeline` takes a heap snapshot and starts recording memory operations until the programmer stops the recording. During the recording interval, `Timeline` does not allow the programmer to interact with the ongoing recordings. After the recording has stopped, `Timeline` generates a timeline in which the programmer can analyze objects over the whole interval or select sub-intervals.

The `Heap Profiler` common window lists the recorded profiles (both snapshots and timelines). For each selected profile, `Heap Profiler` can show memory information in three different views: statistics, containment, and summary. There is a fourth view, called comparison, that is available just for `Snapshot`. Next, we detail the common views and, then, we discuss the comparison view along with other differences between `Snapshot` and `Timeline`.

The statistics view shows a pie chart of the memory usage. It shows the total memory usage divided by the following "types": *code*, *strings*, *JS arrays*, *typed arrays*, and *system objects*. *Typed arrays* are array-like objects that provide a mechanism for accessing raw binary data [50], which is used for websocket manipulation of contents such as video and audio. We could not find a formal description of the view or the types, we assume that that system objects are all other objects that are not in the previous categories.

The containment view shows the objects of the target-program in a very low level. It allows the programmer to look inside function closures, to observe virtual-machine internal objects that together make up JavaScript objects, and to understand how much memory the target-program uses at a low level. This view provides three main entry points: DOMWindow objects, which are considered global objects for JavaScript code; GC roots, which are the actual roots used by the virtual-machine garbage collector; and native objects, which are browser objects that are pushed inside the JavaScript virtual machine to allow for automation, such as DOM nodes and CSS rules. Starting from these objects the programmer can inspect the entire object tree.

The summary view shows basic memory usage information about the target program. The summary view lists all objects grouped by their constructors and, for each object, it shows the shallow size, the retained size, and the shortest distance to the root. The displayed objects can be filtered through a text box where the programmer writes a string that must match part of the constructor's name. Also, the programmer can inspect each object information, such as identifier, context, and reference tree.

The summary view differs between `Snapshot` and `Timeline`. By inspecting a snapshot using the summary view, the programmer can list objects that exist in one snapshot, but do not exist in another. For instance, the programmer can list all objects allocated before `hs1` (heap snapshot 1) that are still alive in `hs2`, or the programmer can list all objects allocated between `hs1` and `hs2` that are still alive in `hs3`. By inspecting a timeline using the summary view, the programmer can list objects that were created during a selected interval.

Comparison is the last `Heap Profiler` view. It is available just for `Snapshot` and allows the programmer to compare two snapshots. The comparison view details memory information of objects that were created or deleted between the compared snapshots.

The Chrome DevTools documentation has a step-by-step tutorial to identify memory bloat using `Heap Profiler`. In this tutorial, the authors present what they call the "3 Snapshot Technique". The methodology used by them was proposed in 1999 by De Pauw and Sevitsky [20]. It is based on the observation that many memory bloats occur during well-defined operations that are supposed to release all of their temporary objects upon completion. In other words, during a program execution, there are different *critical sections* in which all objects that are created inside one critical section should be eligible to reclamation after this section. Therefore, objects created inside a critical section that cannot be garbage-collected right after it cause memory bloat. As an example, imagine a game that has multiple levels. Every new level allocates many objects, and, at the end of the level, all created objects should be eligible to reclamation. If an object allocated during one level cannot be reclaimed at the end of the level, this object is likely causing a memory bloat.

De Pauw and Sevitsky suggest that, given a critical section, the programmer should take a heap snapshot (`hs1`) at its beginning and another heap snapshot (`hs2`) at its end. (Note that both snapshots contain only objects that are not eligible to reclamation). Then, the programmer can compare both snapshots to identify memory bloat, which in this case means objects that exist in `hs2` but do not exist in `hs1`. Once again we will use game development as an example: the programmer should take a heap snapshot at the beginning of the level (`hs1`) and another heap snapshot at the end of level (`hs2`). Then, the programmer should compare both snapshots and identify objects that exist in `hs2` but that do not exist in `hs1`. These objects were allocated during the level execution but are not eligible to reclamation at the end of the level, and, thus, denote memory bloat.

In the "3 Snapshot Technique" tutorial, the authors adapt the critical-section idea to a web browser scenario. In their tutorial, the critical section is a set of actions executed by the user, such as mouse clicks and scrolls.

First, the programmer takes a heap snapshot (`hs1`) and performs the suspicious actions n times (critical section). Then, the programmer takes another heap snapshot (`hs2`) and performs exactly the same actions n times again (another critical section). Finally, the programmer takes a final heap snapshot (`hs3`) and compares the snapshots. Objects allocated between `hs1` and `hs2` that are alive in `hs3` denote memory bloat. Moreover, if these objects appear in multiples of n, the likelihood increases significantly.

Note that in the "3 Snapshot Technique" tutorial the authors leverage the simplicity and predictability of repeating the actions to add a factor n to the number of times the same actions are repeated. This makes it easier to identify objects that are causing memory bloat, as they will be held in memory with a factor of n. Finally, the tutorial uses three snapshots instead of two because each snapshot allocates objects that can obfuscate results. By taking a third snapshot and filtering objects allocated between `hs1` and `hs2`, the programmer eliminates from the report objects created by the snapshot itself.

In 2013, Pienaar and Hundt [23] presented `JSWhiz`, an extension to the open-source Closure JavaScript compiler that helps identifying memory bloat at compile time. The Closure compiler transforms JavaScript code in more efficient JavaScript code. The Closure library is a JavaScript library based on a modular architecture that is written specifically to take advantage of the Closure compiler. The Closure library provides cross-browser functions for DOM manipulations as well as more high-level objects such as user interface widgets and controls. `JSWhiz` focuses on the event system abstraction of the Closure library to detect objects that should be garbage-collected, but are not collected due to some forgotten event reference.

Based on experiences analyzing memory problems in Gmail, Pienaar and Hundt point out common patterns that create useless objects.[1] These patterns are encapsulated within a new concept the authors call eventful class, which is a class that has events associated with it. All patterns are related to event handlers or listeners that, due to different reasons, forbid an object to be garbage-collected. As an example, we can cite one-time event handlers, which are events that are automatically deleted after they are triggered for the first time. When JavaScript programs use these handlers and the corresponding events never occur, listeners are not removed. Accordingly, objects referenced by them are never reclaimed.

In contrast to the memory profilers we discussed before, `JSWhiz` is applied at compile time and, thus, has all the advantages and drawbacks of a

[1]These patterns are very specific to the Closure context and detailing all of them here is not relevant. We discuss the main idea of the paper and highlight pros and cons.

static program analysis. On the one hand, software defects can be discovered earlier, which is excellent as the cost of finding and fixing software problems increases the later the problems are found [23]. Besides, the discovery process is easier as an automated process that executes once is easier than a labor intensive process, such as the "3 Snapshot Technique". Finally, there is no runtime overhead. On the other hand, `JSWhiz` only helps identifying useless objects that arise from the specified patterns. Moreover, it only analyzes variables that have full qualified names (which excludes arrays, lists and other data structures), are returned by a function, and are not captured in closures.

`JSWhiz` found a total of 89 memory bloats across Google's Gmail, Docs, Spreadsheets, Books, and Closure itself. It contributed significantly to reduce Gmail memory usage by roughly 50% at the median. `JSWhiz` compilation time overhead is between 3% and 14%. Also, it executes only in "compile all test cases", which reduces everyday overhead.

2.4 Python Tools

Python has a "batteries included" philosophy [47]. This means that the Python distribution package comes with many modules to make some set of tasks within a particular problem domain simpler. Regarding memory information, Python has two modules to collect data about an executing program: `resource` and `gc`.

The `resource` module provides basic mechanisms for controlling and measuring system resources used by a program. To control system resources, this module offers a function to set the limit of a specified resource and another function to get this limit. As an example, the programmer can set and get the maximum heap size of the executing program. To measure system resources, `resource` offers one single function that returns an array containing the resources consumed by the program. Among others resources, the array contains information about swaps, page faults, shared and unshared memory, and stack. Programmers can use the `resource` module to understand the overall memory consumption of an executing program.

`gc` provides an interface to configure and use some features of the garbage-collector. The `gc` interface allows the programmer to disable the garbage-collector, tune the collection frequency, and force a full collection. It also allows the programmer to set log-levels to print garbage-collection information during collection cycles, such as statistics and objects that will be collected. Finally, `gc` exposes two functions to identify parents and children of given objects: `get_referrer` and `get_referents`. The former returns the list of objects that directly refer to a given object. The later returns the list of objects that are directly referred to by a given object. Both `resource` and `gc` are the base for most of the memory profilers in Python.

`Objgraph` [9], `Heapy` [19], `Pympler` [5], and `Dowser` [25] are manual profilers that let the programmer visually explore Python objects. One can think of them as wrappers to both `resource` and `gc` with some additional information and visual enhancements. The purpose of these memory profilers is to help programmers identify and fix memory bloat in Python programs. The main idea behind these tools is to pick an object that should not be alive and, then, inspect which references are keeping it alive.

Programmers can use `Objgraph`, `Heapy`, `Pympler`, and `Dowser` to count objects, the number of instances for each type, and the names of types with the most instances and to print the types of most common instances. In addition to that, the programmer can check the detailed memory growth by object type since the last check. Also, the programmer can find a shortest chain of references leading from an object `o` to an object `x`. Finally, the programmer can print the complete heap graph.

Memory Profiler [21] is a Python module for monitoring the memory consumption of a program in different ways. Memory Profiler offers three memory profilers in one module: `memory_usage`, `mprofpy`[2], and `memory_profiler`.

`memory_usage` is a manual memory profiler that consists of one function that returns the memory usage of a process (with a given `pid`) over a time defined interval. As an example, the programmer can get the memory consumption of an executing web browser over a period of 60 seconds with a time interval of 1 second. `memory_usage` gets the memory usage by querying the operating system kernel.

`mprofpy` is an automatic and final-report memory profiler that is a wrapper to `memory_usage`. `mprofpy` executes the target-program as a sub-process and monitors this sub-process during its entire execution using `memory_usage`. During execution, `mprofpy` saves metadata into a file. This file is then read by another program that plots the memory usage of the target-program over time.

Finally, `memory_profiler` is an automatic, call-hook, and continuous-report memory profiler. The call-hook is done by annotating the functions that the programmer wants to profile (target-functions) with `@profile`. This annotation uses a Python mechanism called decorator, which changes the call protocol. During the target-program execution, when the target-function is

[2]Although the memory profiler name is `mprof`, we decided to use `mprofpy` to avoid confusion with the original `mprof` that we described in Section 2.2.

```
Line #  Mem usage  Increment  Line Contents
===================================================
   3                          @profile
   4      5.97 MB    0.00 MB  def my_func():
   5     13.61 MB    7.64 MB    a = [1] * (10 ** 6)
   6    166.20 MB  152.59 MB    b = [2] * (2 * 10 ** 7)
   7     13.61 MB -152.59 MB    del b
   8     13.61 MB    0.00 MB    return a
```

Figure 2.1: `mprofpy` output example.

called, Python executes the decorator-function (the function that has the same name as the annotation, which is `profile` in this case) passing as a parameter the target-function and the parameters of the original call. The `profile` function, defined by `memory_profiler`, executes the target function line-by-line and collects memory information before and after each line execution. After the target-function execution, `profile` prints the target-function report.

Figure 2.1 shows the report of an example function called `my_func`. The report is a list of the function lines divided in a four column table. The first column shows the line number, the second column shows the memory usage of the Python interpreter after that line is executed, the third column shows the memory difference of the current line with respect to the previous one, and the fourth column shows the code of the executed line.

As we could not find information about performance or memory overhead, we used one of the tests that came with Memory Profiler to measure overhead. Our simple test showed that `mprofpy` incurs in 1.02x execution time slowdown and 2% memory overhead and that `memory_profiler` incurs in an execution slowdown of 1.7x and a memory overhead of 2%.

2.5 Lua Tools

Although Lua is largely adopted by industry and has many reports regarding memory bloat [34, 36, 35], we could only find four tools that help programmers to understand the memory behavior of a Lua program. In the next paragraphs we detail each one of them.

`Lua Memory Profiler` [40] is an automatic memory profiler written in C++ designed for Lua 5.0. To use it, the programmer needs to modify the Lua interpreter to use the `realloc` and the `free` functions defined by it and recompile the language. The idea used in `Lua Memory Profiler` is very similar to the idea used in both `mprof` and `smallmp`, which we previously discussed in Section 2.2. In summary, one redefines the program allocation function to collect information during memory allocations. Since Lua 5.1, which was released in 2006, the language exposes a function that dynamically sets a given allocation function. However, `Lua Memory Profiler` does not use it.

`ProFi` [22] is small module written in Lua that hooks function calls and function returns to collect execution time data; it was released in 2012. Although `ProFi` focus on execution time, it also has a memory profile option. The memory profile option is simple and consists of a single function called `CheckMemory` that records a pair containing the current memory usage and the elapsed time since the program started. `CheckMemory` can be called as many times as the programmer wants. Each time it is called, a new pair is added to the list. Finally, `ProFi`'s memory-report logs the list of the recorded pairs ordered by time and highlights the maximum and the minimum memory values of the list. One drawback is that both time and memory metadata incurs memory overhead, which is included in the final report and can mask the real memory that was used by the program.

`luatraverse` [31] is a manual memory profiler written in Lua that implements one core function that traverses all references to all live objects of a Lua program and, for each reference, applies a given function; it was released in 2006 and last updated in 2010. Consider a root object `R` that references two objects `X` and `Y`. Then, both `X` and `Y` references `W`, which references `R`. If one calls `luatraverse` passing the print function as parameter, it will print `R`, `X`, `Y`, `W`, `W`, and `R`. `W` is printed twice because there are two references to it, `X` and `Y`. `R` is also printed twice because it is a root object and there is one reference from `W`. Notice that each reference is traversed only once, which avoids cycles. Besides the core function, the `luatraverse` module implements two example functions: `countreferences` and `findallpaths`. The former prints the number of objects that reference a given object. The latter prints all paths to a given object. To execute the core traversal function or any other "enhanced" function such as `countreferences`, the programmer loads the library and adds one line for each "task". As an example, the programmer can add a line before and another after a code chunk for logging all references to a specific object. Then, she can compare the logs to check if there is any unwanted reference after the chunk execution.

`microscope` [13] is a manual memory profiler written in Lua that dumps any Lua value as GraphViz files, which can be later transformed into a nice image of the graph composed by those dumped objects; it was released in 2013. `microscope` consists of a function that receives an object and logs the graph that starts from the given object in the GraphViz format. Different optional parameters can be passed to fine tune the object dump. One example is `nometatables`, which disables logging metatable values. Another example is `size`, which adds size information to logged objects.

2.6 Summary

In this chapter we reviewed multiple papers and tools that explore different techniques to help programmers identify and fix memory bloat. Next we summarize the most important topics to our work.

Regarding general rules, there are three important aspects to take into consideration when developing a memory profiler. One aspect is integration, which concerns how easy it is to use the memory profiler with an existing program. Another aspect is performance, which concerns how much time and memory overhead the memory profiler causes to the main program. Finally, the last aspect is insights, which concerns how easy it is to read and understand the memory reports and, more importantly, how much help these reports provide to identify and fix memory problems.

Regarding techniques, there are seven techniques that we considered promising. We explored five of them to build our proposed memory profilers. However, we did not explore the remaining two techniques due to time and scope constraints.

The first technique we explored consists of collecting memory information about the execution of a program at the function level. It can help users easily understand the flow of the program in terms of function calls. Moreover, this technique can reveal how much data is being allocated by each function, which is particularly interesting to narrow down which parts of the code are potentially causing memory bloat.

The second technique we explored consists of categorizing the allocated objects by type or class. It can help programmers identify data structures that could cause memory bloat. Similarly to the previous technique, this technique also narrows down which parts of the code are potentially causing memory bloat. Programmers should choose whether to use the previous technique or this technique based on the the number of different functions and data structures used by the program. As an example, imagine a program that only uses integer arrays. Using a memory profiler that applies the type categorization to identify potential memory bloats will indicate that the cause of the problem are integer arrays. However, in this case, this information is obvious and, thus, unhelpful.

Memory profilers that explore either continuous-report or final-report techniques are helpful to programmers. Memory profilers that implement continuous-report can have a high impact in performance and memory use. However, they provide information during program execution, which allows the programmer to analyze random parts of the execution and long running programs. Memory profilers that implement final-report usually have a small impact on performance and memory usage. However, the programmer must wait for the entire program execution to analyze the final report. Moreover, the final report provides an overview of the memory allocated during the entire execution instead of more fine-grained information.

The last technique we explored is ongoing interaction with the executing program. It allows programmers to pause program execution to analyze memory information. This technique works similarly to a debugger, but with a better set of methods focused on memory information. As an example, a programmer can pause the execution of a program to investigate why an object is not being garbage collected (i.e. to investigate current references to an object).

The first technique that we did not explore is pretty printers. Pretty printers are interesting tools that can help programmers investigate data structures at specific parts of the code. As an example, the programmer can print an object before and after a complex computation and, then, compare both outputs to verify whether the object holds the expected set of values. However, pretty printers are very limited in identifying which parts of the program contain problems. Identifying which data structure or code blocks one should monitor involves a manual process that can take a long time. In this work, we opted to create automatic tools that can help programmers understand the overall behavior of a program instead of specific parts.

The second technique that we did not explore is static analysis. It is an interesting technique that allows software defects to be discovered early in the software development process, in an automated manner, and with no runtime overhead. However, due to its static nature, static analysis is very limited, especially in scripting languages. Most memory profilers that use static analysis to tackle memory bloat problems rely on predefined patterns. In many cases, to collect enough code samples to analyze and derive these patterns is complicated and takes too much time. Although static analysis can be helpful in dealing with memory bloat and should be used along with dynamic analysis, we have not used this technique.

3
Proposed Tools

Scripting languages are largely used in software development and memory bloat problems have been widely reported in these languages. However, we could find few tools that help programmers with these problems. We believe it is important to analyze and explore techniques to tackle memory bloat in these languages.

To experiment some techniques, we propose two different memory profilers for Lua: `luamemprofiler`, an automatic, allocation-hook, and continuous-report tool; and `lmprof`, an automatic, allocation- and call-hook, and late-report tool.

In this chapter, we first discuss the guidelines used during the design and the development of both tools. Then, we present each tool separately. For each tool, we explain its idea, detail the implementation, and discuss design decisions and development difficulties.

3.1
Guidelines

Regarding profiler goals, we agree with Zorn and Hilfinger [33]. Accordingly, we have developed our tools based on three criteria: a profiler should be easy to integrate into existing programs; a profiler should not impose too much overhead on the target program; and a profiler should provide readable reports for a regular programmer. In the next paragraphs, we discuss the main decisions regarding the design of our memory profilers.

One decision was to implement `luamemprofiler` and `lmprof` as separate tools. Although both tools have some chunks of code that are very similar, and both tools could be easily joined in one big tool, we decided to separate them. By separating them, it was easier to explore the different techniques and analyze their pros and cons. Also, most of the studied tools have different memory profilers for specific purposes.

Another decision was to add only essential features to each tool. In some cases, although it would be easy to implement a new feature, we opted not to add it as it did not seem to fit the tool main purpose. Using `lmprof` as an example, while collecting how much memory a function has allocated, it would be easy to also collect how much time it has executed. However, collecting the execution time was not the main purpose of the `lmprof`. Accordingly, we opted not to add this feature.

Lua is implemented in ANSI C and has a very complete C API to integrate C code with Lua code. Therefore, when creating a library for Lua, one can easily opt to write it in Lua or in C. By implementing it in Lua, one leverages many features, such as automatic memory management, multiple return values, and high level programming. By implementing it in C, one can create faster libraries that consume less memory and that have access to more Lua internal information than if implementing it in Lua. Also, by implementing a library in C, one can allocate memory directly without interfering in the memory used by the Lua interpreter. This is particularly useful to memory profilers as they can track the target-program without worrying that the metadata might influence the report. These trade-offs must be carefully analyzed for each library during the design phase. In our case, as performance is very important, full access to internal information is essential, and separating the amount of memory used by the profiler from the amount of memory used by the target-program is good, we decided to implement both `luamemprofiler` and `lmprof` in C.

In Section 2.1.2, we explained the call-hook and the allocation-hook techniques, which collect memory information during the track phase. Then, for each automatic memory profiler discussed in Chapter 2, we explained how these techniques were implemented. For instance, `mprof` has a special compiler that changes the allocation functions and `memory_profiler` uses the Python decorator mechanism to intercept specific functions. Based on a "mechanisms instead of policies" philosophy, Lua offers two mechanisms to *dynamically* hook Lua programs. One mechanism implements the allocation-hook technique and the other implements the call-hook technique. Below we detail both mechanisms.

The Lua core does all its memory allocation and deallocation through one single allocation function, which the developer must provide when she creates a Lua state [12]. Besides the single allocation function, since Lua 5.1 (released in 2006), Lua exposes a function to dynamically change this single allocation function. The function `lua_setallocf`, which is available just for the C API, allows a new allocation function to be registered during the execution of a Lua program. After setting the new allocation function, all further allocations will use it. Regarding memory profilers, the new function is usually a wrapper around the old one to collect metadata. Both `luamemprofiler` and `lmprof` use this mechanism.

One restriction of dynamically changing the allocation function is that it cannot be used at the same time by two tools that change the allocation function. In this case, the first substitute function would be overwritten by the second and would not record the proper metadata for the first tool. In general, developers only change the allocation function in very specific applications, such as debug tools. Unless the programmer is using more than one debug tool simultaneously, this should not be a problem.

Another restriction of this mechanism is that the new allocation function will only track memory operations that execute inside the Lua state. In other words, Lua tools that are written in C and call `malloc` or `free` directly do not use the new allocation function and, consequently, do not have their data tracked. As all Lua standard libraries do not use C allocation functions directly and the Lua creators suggests that developers use proper Lua functions to allocate data, it should not be a problem.

The second mechanism enables hooks to be set and unset dynamically. The `lua_sethook` function, which is available for both the Lua and the C API, allows the programmer to register a function to be called every time a function is called and every time a function returns. Regarding memory profilers, the registered function can get the memory usage of the program at both events (call and return) to calculate the the memory allocated by each function. The main advantage of the Lua mechanism, compared to the mechanisms used by the tools discussed in Chapter 2, is that it can be turned on and off dynamically. For instance, `gprof` hooks are inserted by recompiling the target-program, `memory_profiler` hooks are inserted by annotating every function that should be profiled, and `Lua Memory Profiler` hooks are inserted by modifying and recompiling the Lua interpreter. `lmprof` hooks, which uses the Lua mechanism, are inserted by a call to `lua_sethook` at any time of the target-program execution and can be set and unset during execution. One drawback of the Lua mechanism is that, as each new hook overwrites the previous one, it cannot be used along with other libraries that also create hooks.

3.2
`luamemprofiler`

Our first proposal is `luamemprofiler`, an automatic, allocation-hook, continuous-report memory profiler. `luamemprofiler` displays the memory allocation of a Lua program in real-time, distinguishes each memory block type by color, and allows the programmer to interact with the program and the displayed information along the program execution. `luamemprofiler` was developed in the Google Summer of Code program [38] in 2011 to explore three main techniques — real-visualization, type/class data categorization, and ongoing interaction.

Real-time visualization can bring a new perspective to memory behavior. For instance, in game creation, one can analyze specific moments where certain data types are created or collected, such as level transitions. Therefore, `luamemprofiler` updates the heap display continuously, so that the programmer can follow memory operations as soon as they occur.

As discussed in Chapter 2, using types/classes to categorize and summarize information can be very useful to understand the memory behavior of a program. Although Lua does not have classes and there are only 8 basic Lua types (nil, boolean, number, string, function, table, userdata, and thread), we wanted to experiment the categorization by these types during profiling. Therefore, each type has a color that is used to draw the memory blocks of the respective type. Unfortunately, as we will discuss later, there are limitations regarding table categorization.

Finally, the last technique provides interactive actions during program analysis. In addition to the real-time display with different types, we wanted to allow the programmer to pause execution, analyze the program heap, and customize the display at runtime. As an example, the programmer may want to analyze just string blocks instead of every block. Accordingly, `luamemprofiler` allows the programmer to pause execution, execute the program step by step, and check the current memory allocation command and its stack trace. Also, `luamemprofiler` allows the programmer to dynamically define which types are going to be drawn and to zoom in and out any heap fragment. Next, we detail the `luamemprofiler` interface and how we use it.

3.2.1 Usability and Interface

`luamemprofiler` is a Lua tool that requires only small changes into the user script. In short, the programmer just needs to add three lines to the original code: one line to load the tool, another one to start profiling, and a last one to stop profiling. If the programmer wants to profile the whole application, she can call `start` at the beginning of the script and `stop` at the end, as exemplified below.

```
local lmp = require"luamemprofiler"
lmp.start(1)  -- 1 is the heap-size-display in MB
-- original script code ...
```

```
04  lmp.stop()
```

The `start` parameter sets the heap size that will be displayed, hereafter called heap-size-display. In the above example, the display will show one megabyte of the heap, even though the program may use more. Choosing the right heap-size-display is not straightforward and we will discuss in the next section. In addition to profiling the whole program, one can also profile specific parts of the same program by adding the start and the stop lines around each part. `luamemprofiler` will create a new display for each part. However, one cannot profile two or more parts at the same time.

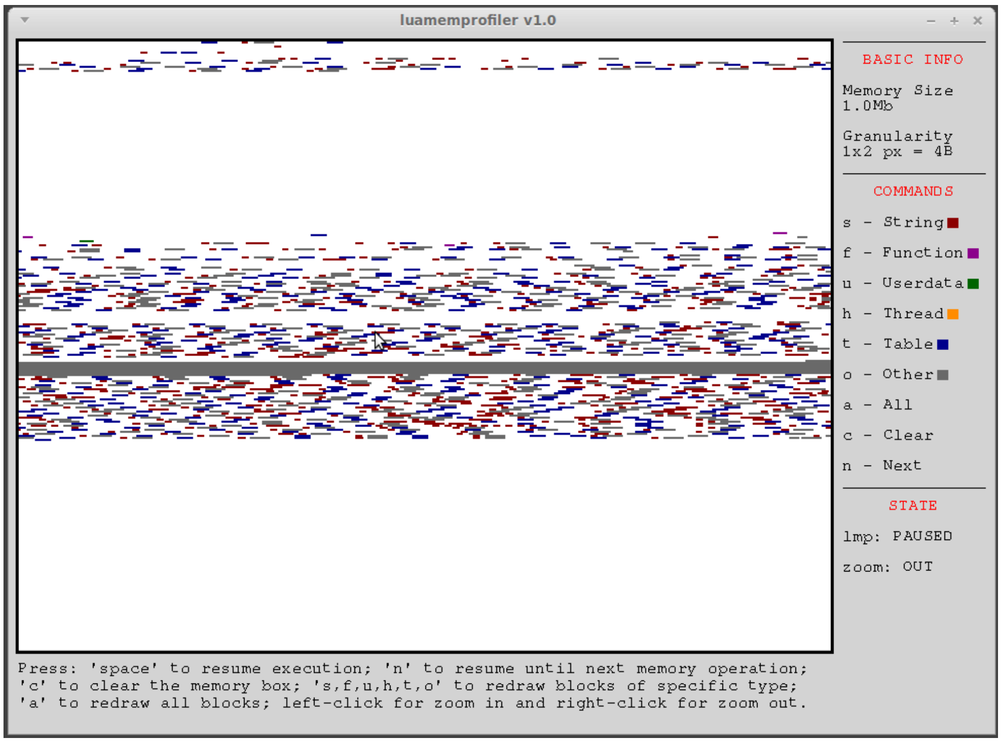
When the `start` function is called, it initializes the `luamemprofiler` module, starts the display, and waits for the programmer interaction. One can press 'space' to run the program or 'n' to execute the program until the next memory operation (`malloc`, `free`, or `realloc`). When the program is executing, all interactions are disabled, except for the 'space' key which pauses the execution and re-enables all interactions. Figure 3.1 shows the display in the paused mode.

At the center of the image is a snapshot of the heap as a two dimensional picture in which each pixel is associated with a memory location starting at the top left-hand corner. The "Memory Size" is the heap-size-display and means how many bytes of the heap we are drawing; in this case 1 megabyte. Unused memory locations are left blank. Blocks are represented by horizontal stripes. That is, memory blocks are displayed line-by-line from left to right with the granularity indicated at the top right-hand corner. The larger a block is, the longer is the associated stripe. Also, parts of the heap can be magnified.

Blocks are distinguished by their color. There is a list of the Lua types with their respective colors and the keys used to toggle (*select* or *unselect*) each one of them. For instance, all strings are displayed in red and can be toggled by the letter 's'. Every time a type is selected or unselected, the blocks of that type are drawn or erased from the display. Instead of 8, there are only 5 basic Lua types indicated. This is a limitation that we will discuss in the next section.

There are two possible program states: paused and executing. If the program is in paused state the programmer can do a step by step execution. In this case, `luamemprofiler` displays at the bottom the current operation information followed by its stack trace, as shown below.

```
Malloc | addr = 0x241b9901 | type = String | size = 96B
C - func'for iterator'
Main - line:86
```


luamemprofiler v1.0

BASIC INFO

Memory Size
1.0Mb

Granularity
1x2 px = 4B

COMMANDS

s - String 🟥
f - Function 🟪
u - Userdata 🟩
h - Thread 🟧
t - Table 🟦
o - Other ▨
a - All
c - Clear
n - Next

STATE

lmp: PAUSED

zoom: OUT

```
Press: 'space' to resume execution; 'n' to resume until next memory operation;
'c' to clear the memory box; 's,f,u,h,t,o' to redraw blocks of specific type;
'a' to redraw all blocks; left-click for zoom in and right-click for zoom out.
```

Figure 3.1: `luamemprofiler` graphical display.

In addition to the real-time display, every time `stop` is called, `luamem-profiler` logs a summary of the heap allocations. Figure 3.2 shows the final report of the SparseMatMult application used for evaluation in Chapter 4. We can see that the main program performed 192 malloc operations, 302 realloc operations, and 25 free operations. Also, these operations allocated 17.8 kilobytes, reallocated 400 megabytes, and deallocated 10.5 kilobytes. Below the free information, we summarize the mallocs by type. Figure 3.2 shows that from the 192 mallocs, 60 were strings, 11 were functions, and so forth. Then, the maximum memory used is the memory-usage peak of the program. Although the example allocated more than 417 megabytes along the program execution (the malloc size plus the realloc size), the maximum memory occupied by the program was 400 megabytes. Finally, at the last line `luamemprofiler` suggests the minimum heap-display-size that should be used as a parameter to `start`. This value is the difference between the biggest and the smallest block-address of the whole execution in megabytes. Although sometimes this value is very similar to the maximum memory used, it can be very different depending on the allocation policy.

From all the values presented in the final report, the `total realloc size` is the only one that is not straightforward. It is the sum of the difference between the new and the original blocks of all the `realloc` calls. In other words, every time `realloc` is called, `luamemprofiler` adds to the `total realloc`

```
Number of Mallocs  = 192      Total Malloc Size  =  17.8 Kb
Number of Reallocs = 302      Total Realloc Size = 400.0 Mb
Number of Frees    =  25      Total Free Size    =  10.5 Kb

Number of Mallocs of Each Type:
  String = 60 | Function = 11 | Userdata =   0
  Thread =  0 | Table    = 20 | Other    = 101

Maximum Memory Used = 400.0 Mb

We suggest 410.3 as heap-size-display parameter.
```

Figure 3.2: `luamemprofiler` final report of the SparseMatMult test. The format has been slightly altered to better fit this thesis.

`size` the difference between the new and the original block, which is a positive value if the block is expanding and a negative value if the block is shrinking. In Figure 3.2, it is obvious that `realloc` is responsible for manipulating a lot of data. However, sometimes the current definition can mask how much data was manipulated by `realloc` calls. Accordingly, in a future release, we should separate expanding reallocations from shrinking ones.

3.2.2
Trade-Offs

During `luamemprofiler` design and development, we made some decisions and had a few problems that we consider worth mentioning in this thesis.

Our first design decision was that regardless of the graphical library used in the original implementation, the graphic part should be independent and easy to change. Therefore, we designed the graphic part as a separate layer and defined an interface that can be implemented using different graphical tools.

In addition to the flexibility in the graphical library implementation, we added a flag to use or not the real-time display. The programmer may be interested in the final summary, or in the heap-size-display suggestion. Moreover, the real-time display can cause performance penalty, which may be a problem in some cases. In order to disable the display, `start` must be called without parameters.

One problem we had in the beginning of the tool implementation was a segmentation fault error at the end of every target-program execution. That is, every target-program was executed up to the last line without errors and `luamemprofiler` returned correct results, but there was a segmentation fault in the Lua finalization process. This problem occurrs because, at the end of a program execution, Lua calls the allocation function to free some structures.

As `luamemprofiler` is garbage-collected, the allocation function it uses is also reclaimed, which causes errors in future calls. As other programming languages, Lua offers a mechanism for object finalization. This mechanism sets a function to be called after an object is collected. `luamemprofiler` takes advantage of this feature and sets a finalizer for the library object that restores the original allocation function.

During `luamemprofiler` development, we had two problems regarding the categorization of blocks by types. `luamemprofiler` tracks allocated blocks using the allocation hook explained in Section 3.1, which dynamically changes the allocation function. Trying to facilitate tools such as memory profilers, when a new block is allocated, Lua passes the block type to the allocation function.

The first problem is that `luamemprofiler` does not have the block type information for `free` and `realloc` operations. The `free` information would be important to the programmer in the final report to discover types that are not being garbage-collected, by comparing to the `malloc` information. The `realloc` information is particularly important to understand the memory behavior of Lua programs because, differently from many systems that have automatic memory management, Lua extensively uses `realloc` to allocate new objects. This is a way to organize memory without creating garbage. Accordingly, `luamemprofiler` does not have the block type information of many new blocks.

The second problem is that the type information does not work well for *tables*. Tables are the sole data-structuring mechanism in Lua; they can be used to represent ordinary arrays, sequences, symbol tables, sets, records, graphs, trees, etc. Therefore, most of the data of a Lua program is usually inside tables. Tables are represented as a header that describes the table information and a body that holds the table content (which is the majority of the data). When Lua creates a new table, it allocates both the header (which has table type) and the body (which has undefined type). We use *other* to reference all blocks that do not have a specific type. Consider the following code that creates a new table and then inserts the value 10 at the key 1.

```
01 local t = {}
02 t[1] = 10
```

Lua will allocate the table header at line 1 (table type) and the table content at line 2 (*other* type). The value and key numbers are incorporated into the content allocation. In practice, `luamemprofiler` reports have many blocks allocated with type *other*, which is not useful to detect memory bloat.

Finally, the last problem we had during `luamemprofiler` development was the graphic part. The major problem is how to display the complete heap of a program. A poor solution is to use a very large heap-size-display. In this solution, programs that use little memory will be badly displayed. Also, there is no guarantee that a new allocated block fits in the display. Finally, unless there is a scroll bar, blocks will be very small in the screen and will likely collide with others (same pixel will represent different blocks). Another solution is to start with a small heap and redraw the screen when a new block is allocated outside the display. Redrawing the screen solves all problems above, as the screen will display almost the same size as the heap. However, it can perform badly as new blocks are allocated outside the display (even if we consider a display resize offset). The last option is the perfect scenario, in which the programmer knows the complete heap size ahead execution. In this case, all blocks will fit and there will be no redraw. Unfortunately, one usually does not know the heap size of a program and, moreover, from one execution to another the heap size can differ. We tried to join simplicity and precision by implementing a heap size suggestion, previously explained. Our implementation defines the heap size from start and ignores all blocks that do not fit the heap display. We believe that these few "missing" blocks are not essential to understand the allocation behavior of a program. Unfortunately, the programmer usually depends on the `luamemprofiler` suggestion which requires a first program execution. This first execution can be a problem in long running programs.

3.3 `lmprof`

Our second proposal is `lmprof`, an automatic, allocation- and call-hook, final-report memory profiler. Similarly to `mprof`, `lmprof` tracks function calls that allocate data and, after program execution, summarizes the information to the programmer. `lmprof` was developed in the Google Summer of Code program in 2014 to explore two techniques — function profiling and final-report.

Profiling a program at the function level is very common and helps programmers understand the work flow of the program and identify bottlenecks. In languages in which the memory allocation is performed explicitly, programmers are often aware of the functions that allocate memory. However, in scripting languages such as Lua, memory allocations happen implicitly all the time and, thus, programmers are not aware of memory consumption, moreover at the function level. Accordingly, we believe that this approach generates useful information for programmers.

To track a program with `lmprof`, the programmer adds the same lines that are added to use `luamemprofiler`, except by the `start` parameter that is not needed.

```
local lmprof = require"lmprof"
lmprof.start()
-- original script code ...
lmprof.stop()
```

During `start`, a hook is set to intercept all functions. Then, when a function is called, `lmprof` records the total allocated memory. When this function returns, `lmprof` checks if any data was allocated during the function execution (more specifically, if the total allocated memory is bigger than the total allocated memory recorded at the function call). If true, `lmprof` increments the call counter, the shallow memory size, and the retained memory size of the current function. Finally, when `stop` is called, `lmprof` saves the metadata into a file in the form of a Lua table.

The report phase of `lmprof` is a separate program executed after the target-program execution. It reads the output file, consolidates allocation information and prints two profiles: *flat* and *call-graph*.

The flat profile consists of a list of all the functions that allocate data during program execution and how much data they allocated. It gives a quick overview of the functions that are used, and shows the functions that are themselves responsible for large fractions of the allocated memory.

Figure 3.3 shows the flat profile of the SparseMatMult application used for evaluation in Chapter 4. The first line indicates that only three of ten functions that allocated data are listed. Then, each function is detailed line by line in decreasing order of shallow memory. From left to right the details are the percent of total allocation that took place in each function, the number of bytes allocated by each function alone, the number of bytes allocated by the function including its descendants, the number of calls made to the function that resulted in memory allocation, the mean number of bytes allocated by each function call alone, the mean number of bytes allocated by each function call including its descendants, and the function name. In this example, `newvec` at line 29 of the file `SM.lua` allocated 97.99% of the memory allocated by the program, which corresponds to 392 megabytes, in 12 calls. RandomVector was responsible for 2% and, after that, no other function was responsible for more than 0.01% of memory allocation. The last function listed is `dofile`, which is actually a C code function.

Understanding the memory allocation behavior of a program sometimes requires more information than just knowing the functions that are directly responsible for memory allocation. The call-graph can be seen as a call-chain

```
======= Showing 3 of 10 functions that allocated memory ======

    %     shallow  retained          shallow   retained
   mem      mem       mem    calls  mem/call  mem/call  name

  97.99  392.0 MB  392.0 MB    12   32.67 MB   32.7 MB   nv*
   2.00    8.0 MB    8.0 MB     1    8.00 MB    8.0 MB   RV*
   0.01    0.0 MB    0.0 MB     1    0.02 MB    0.0 MB   df*

* newvec (SM.lua:29) | RandomVector (SM.lua:80) | dofile [C]
```

Figure 3.3: `lmprof` flat profile of the SparseMatMult test. The format has been slightly altered to better fit this thesis layout. `nv*` refers to `newvec (SM.lua:29)` and so forth.

in text format where the functions with more retained memory are the main nodes. The call-graph shows all the functions that were indirect callers of functions that allocated memory.

Figure 3.4 shows an example of the call-graph profile. Below the header, each row can be divided into three parts: the dashed line for the function itself, here called entry function; lines above that line, each of which represents a caller of the entry function (the parents); and lines below that line, each of which represents a function called by the entry function (the children).

The major entries of the call-graph profile are the entries from the flat profile, augmented by the memory propagated to each function from its descendants. Regarding parents and children, the number of calls is relative to the respective entry function. For instance, the third row shows that `f4` was called 5 times by `f2`, 2 times by `f3` and 5 times by itself (we use parenthesis to indicate recursive calls). The same applies to `shallow` and `retained`. `f4` allocated 50 megabytes when called by `f2` and 20 megabytes when called by `f3`. The `index` field provides a unique index to help users navigate through the call-graph. Different from the flat profile, the call-graph profile is ordered by the retained memory size.

3.3.1 Trade-Offs

In the next paragraphs, we highlight design decisions and discuss major problems we had during development.

To collect the memory information of a program in a function level, a memory profiler can use the allocation-hook or the call-hook techniques. By using the allocation-hook technique, for every allocation, the profiler can get information about the current and the parent function. By using the call-hook

```
                                   call/total  parents
index   %mem   retained   shallow  call(rec)   name       index
                                   call/total  children
===============================================================
[1]----100.00--160.0-MB----0.0-MB------1---------f1-------[1]
                130.0 MB   80.0 MB     2/2        f2       [2]
                 30.0 MB   10.0 MB     1/1        f3       [4]
===============================================================
                130.0 MB  130.0 MB     2/2        f1       [1]
[2]-----81.25--130.0-MB---80.0-MB------2---------f2-------[2]
                 50.0 MB   50.0 MB     5/12       f4       [3]
===============================================================
                 50.0 MB   50.0 MB     5/12       f2       [2]
                 20.0 MB   20.0 MB     2/12       f3       [4]
[3]-----43.75---70.0-MB---70.0-MB-----12(5)------f4-------[3]
===============================================================
                 30.0 MB   10.0 MB     1/1        f1       [1]
[4]-----18.75---30.0-MB---10.0-MB------1---------f3-------[4]
                 20.0 MB   20.0 MB     2/12       f4       [3]
===============================================================
```

Figure 3.4: `lmprof` call-graph profile of the SparseMatMult test. The format has been slightly altered to better fit this thesis layout.

technique, on every function call, the profiler can push into a stack the *total* allocated memory and, on every function return, the profiler pops the value that was pushed when the current function was called and compare it to the new *total* allocated memory. If the current value is bigger than the pushed one, it means that an allocation happened and, thus, the profiler needs to collect information about the current and the parent functions.

`lmprof` uses both hook techniques. We initially opted to use only the call-hook technique to explore a Lua mechanism different from the allocation-hook used in `luamemprofiler`. However, the first implementation was collecting the *current* allocated memory, which does not work. Imagine that the *current* allocated memory is 150 bytes when a function is called. Then, this function allocates 100 bytes and, before it returns, the garbage-collector executes and deallocates 200 bytes. When the function returns the current allocated memory is 50 bytes, so `lmprof` cannot determine if the function allocated any block. `lmprof` uses the allocation-hook technique to solve this problem. `lmprof` substitues the default allocation function for a very similar function that increments an allocated-bytes counter every time memory is allocated (both `malloc` and `realloc`). Finally, `lmprof` can collect the *total* allocated memory to check whether there was any memory allocation between a function call and return.

Although `lmprof` hooks all function calls, it only profiles function calls that generate allocation (shallow or retained). Therefore, the report does not have the total number of function calls, but the total number of function calls that allocated memory (directly or indirectly). On the one hand, counting all function calls can be useful to better understand the program workflow. On the other hand, counting all function calls seems useless to analyze the memory behavior of a program. In comparison to other tools, while `gprof` counts every function call, `mprof` counts only function calls that allocate data. Although from the code perspective it is very easy to include this information, we did not include it in the first release because it is not directly related to memory consumption.

During profiling, there are some computations that the developer can choose to execute during the track phase or the report phase. Developers usually choose the report phase to diminish the overhead of the track phase. However, sometimes it is way more complicated to do the computation in the report phase rather than in the track phase. We believe that these cases should be carefully analyzed, and that tests must be done to evaluate the overhead caused. In `lmprof`, we decided to calculate the retained size information in the track phase. The computation of the retained size could be done solely in the report phase. In this case, based on the entry function and its parent, `lmprof` could traverse the call-graph and calculate the retained size for each function. However, this algorithm is not trivial and must handle cycles. We opted for the track phase because it was easier for us to implement and tests presented an overhead of less than 1% to the execution time and less than 2% to the memory consumption.[1]

An interesting feature of functions in Lua is that Lua does *tail-call elimination* [12]. A tail call happens when a function calls another as its last action, so it has nothing else to do. In such situations, the program does not need to return to the calling function when the called function ends. Therefore, after the tail call, the program does not need to keep any information about the calling function in the stack. The Lua interpreter takes advantage of this fact and actually does not use any extra stack space when doing a tail call.

In Lua, as tail calls do not return control to the parent function, they cannot be handled by a profiler in the same way as normal function calls. Instead of collecting the metadata of a function `foo` in its return hook, the profiler should collect `foo`'s metadata in the tail-call hook of the function that is tail called by `foo`. Imagine a function `f1` that calls `f2`, which calls `f3` and

[1]We used the applications from Chapter 4 to compare an implementation that calculates the retained size at the track phase to an application that calculates the retained size at the report phase.

```
| function | mem-size | shallow-size | retained-size |
  ret foo      10b            X               X
    foo         8b     2b (10b - 8b)   2b (10b - 8b)
    foo         6b     2b ( 8b - 6b)   4b (10b - 6b)
    foo         4b     2b ( 6b - 4b)   6b (10b - 4b)
    foo         2b     2b ( 4b - 2b)   8b (10b - 2b)
    main        0b     2b ( 2b - 0b)  10b (10b - 0b)
```

Figure 3.5: Pseudo call-stack of an example program with recursive cycles.

“tail calls” `f4`. `lmprof` collects the metadata about `f1` $\rightarrow$ `f2` when `f4` is “tail called”. Also, when `f4` is “tail called”, `lmprof` saves `f2`’s reference in a variable, so that the metadata about `f2` $\rightarrow$ `f4` is correctly collected when `f4` returns control to `f1`. If there are multiple tail calls in sequence (e.g. `f4` tail calls `f5`, which tail calls `f6`), `lmprof` applies the same process for each tail call.

Cyclic-function calls can be a big problem while profiling, as they introduce spurious allocation relations [33]. Cycles happens due to recursive functions or functions that call one another (directly or indirectly during program execution). Next, we describe a typical problem with recursive functions.

Consider a program with a `main` function that calls a `foo` function, which calls itself three times. Imagine that each function call allocates 2 bytes, so the memory allocated by program is 10 bytes. Figure 3.5 illustrates a pseudo call-stack of the program execution when the last call to `foo` is returning. From left to right one can see the function name, the total memory size at the moment the function was called, the memory allocated by the function itself (memory size of the next call minus the memory size of the function call), and the memory allocated by the function including descendants (memory size of the program minus the memory size of the function call).

Analyzing the allocation size of each caller/callee pair, the result is that *System* $\rightarrow$ `main` allocated 2 bytes shallow and 10 bytes retained, `main` $\rightarrow$ `foo` allocated 2 bytes shallow and 8 bytes retained, and `foo` $\rightarrow$ `foo` allocated 6 bytes shallow and 12 bytes retained. That result is obviously wrong, as the whole program allocated 10 bytes. The problem is the propagation of the retained value, which is accounted twice (for both the entry function and the parent function) in all recursive calls.

To handle this problem, both `gprof` and `mprof` adopt, what they call, the most conservative solution. They discover, during the report phase, strongly-connected components in the call-graph and treat each such component as a single node. In that case, the pseudo call-stack would be as in Figure 3.6. As the cycle is removed, the retained size of `main` $\rightarrow$ `cycle` is 8 (which is correct). However, the function-calls executed inside the cycle becomes obscure.

```
| function | mem-size | shallow-size | retained-size |
  ret cycle    10b          X               X
    cycle       2b     8b (10b - 2b)   8b (10b - 2b)
    main        0b     2b ( 2b - 0b)  10b (10b - 0b)
```

Figure 3.6: Pseudo call-stack of an example program with recursive cycles treated as a single component.

`lmprof` uses a different method to handle cycles. It checks if both the caller and the callee are the same function and, if true, it records just the callee `shallow-size` (without accounting the `retained-size`). In that implementation, the pseudo call-stack would be equal to the original, except by the `retained-size`. The `retained-size` would have zero bytes in the top three calls of `foo` because the parent function is also `foo`. In this method, the call-graph is clear and no function is omitted. However, this method cannot handle cycles between different functions. We opted for this method because we believe that it is easier to implement and that the fine-grained information is more important than handling different function cycles (which we consider a rare case).

4
Evaluation

In this chapter we evaluate both `luamemprofiler` and `lmprof` in practice. We evaluate our tools based on the three criteria discussed in Section 3.1. First, we discuss how easy it is to incorporate each tool into the target program and to generate reports. Then, we analyze how each tool can help programmers understand the memory behavior of a Lua program, Finally, we present the execution time and the memory overhead of each tool.

In order to evaluate these three aspects, we used six different applications developed by third party programmers.

Black and Scholes (BAS) — a financial application, ported from the PARSEC [2] benchmark suite, that calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation.

CAPTCHA JPEG Filter (JPG) — an application to filter CAPTCHA [1] images in the JPEG format to make it easier to perform automatic optical character recognition (OCR).

CAPTCHA PPM Filter (PPM) — an application similar to the above that filters portable pixmap format (PPM) images. While the previous application uses a C library to manipulate JPEG files, the PPM application does all the computation (using strings) inside Lua.

Series (SRS) — a numerical application, ported from the Java Grande [27] benchmark suite, that calculates the first N Fourier coefficients of a function.

SparseMatMult (SMM) — a numerical application, also ported from the Java Grande benchmark suite, that uses an unstructured sparse matrix stored in compressed row format with a prescribed sparsity structure.

[1]CAPTCHA stands for "Completely Automated Public Turing test to tell Computers and Humans Apart". CAPTCHA images show distorted texts that users must type to prove they are humans to a computer system.

Placco (PLC) — an application created at PUC-Rio to transform publication data of the university from the internal format to the format used by the Brazilian government.

4.1 Ease of Use

To profile a Lua program with `luamemprofiler` or `lmprof`, the programmer adds two lines at the beginning of the main file and one line at the end of the same file. Besides doing it manually, the programmer can create a simple wrapper to include these lines automatically. As an example, to execute each test application, we created a simple shell script that, before executing each test, creates a copy of the application containing these three lines, executes this copy, and removes it.

Both `luamemprofiler` and `lmprof` generate a file at the end the application execution. `luamemprofiler` generates a file containing the final report, which can be read with any text editor. `lmprof` generates a file with the metadata as a Lua table. To generate `lmprof`'s final report, the programmer has to execute a Lua script that comes in the `lmprof` package. The Lua script expects two arguments, the report type (flat, call-graph, or both) and the path of the file that contains the metadata. During execution, the Lua script consolidates the metadata and prints the respective report.

The current `luamemprofiler` implementation has a drawback. `luamemprofiler` uses SDL [37] as its graphic library and SDL_ttf to write text in the screen. To load the font file we defined a specific path in the source code, which was set to the current path. Accordingly, to profile the tests with `luamemprofiler`, we had to copy the font file to each test folder. Next, we compare our tools to the automatic tools studied in Chapter 2.

`Heap Profiler`, discussed in Section 2.3, is the most complete and easy to use tool. The entire tool is built on top of Chrome and uses advanced graphical features. Moreover, everything is done via mouse clicks, including graph navigation. This is a very particular case because, differently from other scripting languages, JavaScript is mostly used to create web applications that execute inside browsers and, thus, JavaScript profilers can rely on advanced graphical frameworks.

When compared to the publisehd work studied in Section 2.2, the effort to incorporate the memory profiler into the application is different. All the automatic tools that we discussed recompile the applications, which sometimes can be a problem. However, these tools do not need to change the application source code, which is good for programmers. The reports that our tools generate are quite similar to the reports studied.

Regarding Python, to use `mprofpy` the programmer just needs to modify the command line that executes the script, which is very easy. It is also very easy to generate the final report from the metadata. Another Python tool is `memory_profiler`. To use it, the programmer has to annotate every function, which requires more work than just loading and starting a tool. Also, `memory_profiler`'s report is continuously printed in the standard output, which can be confusing if the application also prints information.

By doing small changes in both `luamemprofiler`'s and `lmprof`'s source code, our tools would automatically start profiling when loaded and automatically stop profiling at the end of the application execution. In that case, the programmer would incorporate the memory profiler via command line, without modifying the application code. One drawback of this implementation, is that if the programmer wants to profile just specific parts of the target program, she will have to call `stop` right after loading the library.

4.2 Report Usefulness

In this section, we analyze how useful the reports generated by `luamemprofiler` and `lmprof` are. We do that by using both tools to understand the memory behavior of two of the six explained applications. Besides the application overview and input, we have deliberately no information about the application implementation or its source code. Accordingly, we use our tools to extract as much information as we can with just a basic idea of each program. After analyzing the reports, we compare our findings to the source code and try to identify and remove memory bloats.

4.2.1 Black and Scholes

Black and Scholes works with portfolio prices and uses a synthetic input, provided with PARSEC, that is based on the replication of 1,000 real options. The input consists of a structured ASCII file about an option, where each line provides information divided in nine columns. To analyze the application, we used an input containing 1 million lines with approximately 63 megabytes.

Figure 4.1 shows the `luamemprofiler` final report. We can see that `malloc` was called 8 million times to allocate approximately 1.14 gigabytes and that only 6 million blocks were deallocated, summing up approximately 600 megabytes. Therefore, at the end of the execution, the application holds

```
Number of Mallocs  = 8001694    Total Malloc Size  = 1.14 GB
Number of Reallocs =     162    Total Realloc Size = 0.13 GB
Number of Frees    = 6000821    Total Free Size    = 0.61 GB

Maximum Memory Used = 0.68 GB

Number of Allocs of Each Type:
  String = 1001662 | Function = 1000004 | Userdata =       2
  Thread =       1 | Table    = 1000009 | Other    = 5000016
```

Figure 4.1: `luamemprofiler` final report of the Black and Scholes application. The format has been slightly altered to better fit this thesis layout.

valid references to almost 2 million blocks that sum 540 megabytes, which is more than 40% of the allocated memory. Also, the maximum memory used indicates that the application holds a maximum of 700 megabytes at the same time. Based on the application overview (price calculation) and the input size (61 megabytes), we should verify if all this data is needed until the end of the execution.

By analyzing the block type information, we can see that strings, functions and tables are allocated 1 million times and that *other* is allocated 5 million times. As the input is a file with 1 million lines, we suspect that every line generates an interaction that allocates one function, one table, and five *other*. This behavior seems reasonable and indicates that we should look into each iteration to check if the blocks that are kept should be deallocated.

The graphical display shows that the application can be divided in three phases. In the first phase, which executes for a short period, the application allocates and deallocates many blocks. Also, there are frequent garbage collections that remove a lot of blocks. At the end of this phase, there are still many blocks in memory. The second phase executes for a very long time. During this phase, there is a very CPU intensive computation, and there is no allocation or deallocation of blocks. Finally, in the third phase, which executes for a very short period, the program does just a few memory operations. By using the graphical display, we could identify that the first phase is where the memory is really allocated. Accordingly, we should analyze just the code of the first phase, which means analyzing 40 lines of code, instead of 150.

Figure 4.2 shows the `lmprof` flat report. The first line indicates that there are 699 functions that allocate data. Then, we can see that most part of the allocated memory (77.20%) is done inside the `main chunk` and that three other functions also stands — `insert` (8.65%), `for iterator` (7.01%), and `gmatch` (5.89%). Among other calls to `insert`, 147 calls allocated memory.

```
===== Showing 5 of 699 functions that allocated memory =====
   %     shallow   retained
  mem      mem        mem        calls  name
 77.20    999 MB    1295 MB          1  main chunk (main.lua)
  8.65    112 MB     112 MB        147  insert [C]
  7.01     91 MB      91 MB    1000001  for iterator [C]
  5.89     76 MB      76 MB    1000000  gmatch [C]
  1.24     16 MB      16 MB          1  ? (main.lua:194)
```

Figure 4.2: `lmprof` flat report of the Black and Scholes application. The format has been slightly altered to better fit this thesis layout.

This is similar to the number of reallocations, which makes sense as inserting a new pair into a table can force the system to reallocate the table. We can also see that both the `for iterator` and the `gmatch` functions are called by the main loop of the application and allocate data on every call. This behavior also makes sense, as the iterator is probably used to read each line and `gmatch` is probably used to read each value of the line. Moreover, the memory allocated by each function is close to the file size. Therefore, we should investigate the main chunk to understand what is allocating almost 1 gigabyte.

The `lmprof` call-graph report was not very useful to analyze the Black and Scholes application. Therefore, we do not detail it here.

After the profile analysis that we described above, we investigated the application source code A.1. We started by investigating the first phase, which is composed by two separate iterations. The first iteration reads each line of the input file, breaks it into 9 different strings, inserts all the strings into a table `t1`, and inserts `t1` into a table `t0`. The second iteration copies the entire data of `t0` into six other tables.

We could identify three problems in the original implementation. The first problem is that `t0` is never used again after the first phase. Therefore, the programmer should assign *nil* to `t0`, so that the garbage-collector can reclaim the table. After we assigned nil to `t0`, the `free` values of the `luamemprofiler` final report increased to almost the same values as the `malloc`.

The second problem is that there is no need to use `t0`. The programmer can iterate the input file and insert the data directly from `t1` into the six tables. After this modification, we reduced the `maximum memory used` measured by `luamemprofiler` from 700 megabytes to 288 megabytes. Also, the memory-usage measured by the time command was reduced from 783 megabytes to 320 megabytes.

Finally, the last problem is that, for each line of the input file, the application creates a new table, assigns it to `t1` and inserts 9 values into

```
Number of Mallocs  = 2006924   Total Malloc Size  = 167.40 MB
Number of Reallocs =     142   Total Realloc Size = 112.00 MB
Number of Frees    = 2006866   Total Free Size    = 167.40 MB

Maximum Memory Used = 192 MB

Number of Allocs of Each Type:
  String = 1006888 | Function = 1000004 | Userdata =  2
  Thread =       1 | Table    =       9 | Other    = 20
```

Figure 4.3: `luamemprofiler` final report of the "fixed" Black and Scholes application. The format has been slightly altered to better fit this thesis layout.

the table. Accordingly, every iteration allocates new blocks that will become garbage in the next iteration. Instead of creating a new table for each line, the programmer should create the table once and overwrite the values on each iteration. After this last modification, the number of allocated blocks dropped from 8 million to 2 million and the memory-usage measured by the time command dropped from 783 megabytes to 190 megabytes. Figure 4.3 shows the `luamemprofiler` final report for the modified version of the Black and Scholes application A.2.

4.2.2 CAPTCHA JPEG Filter

The CAPTCHA JPEG Filter is an application that applies different filters to a CAPTCHA image to make it easier to process the text contained in the image with optical character recognition (OCR). It uses a pipeline to apply filters sequentially to images. The application applies the following filters, respectively:

1. grayscale, which converts the image colors to a range of shades of gray, preparing it for the next filters;

2. binary threshold, which converts the image colors to either black or white according to the brightness of each pixel, to eliminate noise;

3. gaussian blur, which clouds the image and makes it appear as if it is viewed through a translucent screen, to reduce detail;

4. binary threshold, same as above, applied a second time to eliminate more noise;

5. invert, which converts black to white and white to black, to change contrast.

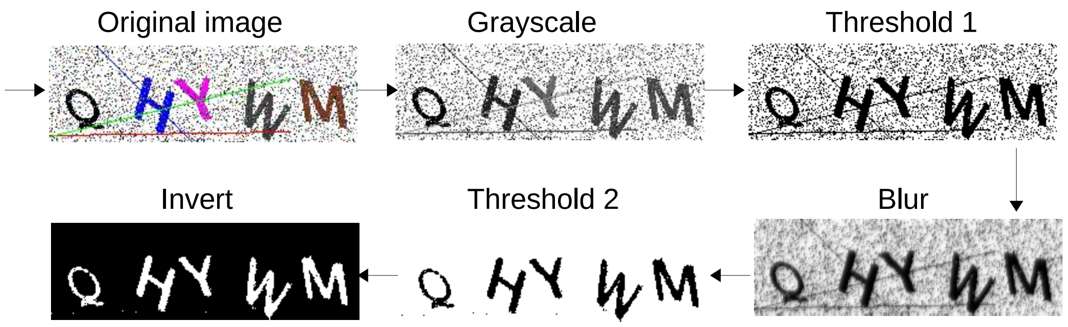


Figure 4.4: Sample results from sequentially applying each of the filters implemented in the CAPTCHA Filter application.

```
Number of Mallocs  = 76790     Total Malloc Size  = 679.07 MB
Number of Reallocs =   294     Total Realloc Size =   0.13 MB
Number of Frees    = 71534     Total Free Size    = 678.86 MB

Maximum Memory Used = 1.36 MB

Number of Allocs of Each Type:
  String = 40970 | Function =  7 | Userdata = 35001
  Thread =     0 | Table    = 10 | Other    =   802
```

Figure 4.5: `luamemprofiler` final report of the CAPTCHA JPEG Filter application. The format has been slightly altered to better fit this thesis layout.

Figure 4.4 shows an example of how an input image looks like after passing each of the filters in the applications. To analyze the application, we used an input of 5000 images with approximately 13 kilobytes each and 65 megabytes total.

Figure 4.5 shows the `luamemprofiler` final report. We can see that `malloc` was called 76000 times to allocate approximately 680 megabytes and that most allocated blocks were deallocated. Moreover, the maximum memory used indicates that the application holds a maximum of only 1.36 megabytes. The above indicates that the application generates many blocks that are frequently garbage-collected.

The block type information shows that the application has few tables with few fields (*other*). Also, the most used types are string and userdata. We suspect that both string- and userdata-blocks are related to reading the JPEG files and converting them from userdata to string and the other way around.

The flat-report supports our suspicion. As shown in Figure 4.6, there are two C functions that are responsible for more than 99% of the allocations. The `read` function is part of the Lua API and is used to read files in text mode. It is probably responsible by part of the string allocations. The `jpegStr` function is probably responsible for converting userdata objects that represent JPEG

```
===== Showing 2 of 24 functions that allocated memory =====
   %     shallow   retained
  mem     mem        mem      calls  name
 78.93  536.6 MB   536.6 MB   25000  jpegStr [C]
 20.64  140.4 MB   140.4 MB    5000  read [C]
  0.17    1.1 MB     1.1 MB   25000  createFromJpegStr [C]
```

Figure 4.6: `lmprof` flat report of the CAPTCHA JPEG Filter application. The format has been slightly altered to better fit this thesis layout.

images into strings. Accordingly, the `createFromJpegStr` function is probably responsible for converting strings that represent JPEG images into userdata objects. As `jpegStr` allocates almost 80% of the memory, we should investigate if it is necessary to convert userdata images into string images and back again.

Finally, the call-graph report in Figure 4.7 confirms that we should investigate both `jpegStr` and `createFromJpegStr`. The first and the third rows show that there are 5000 calls that allocate data to each filter-function (threshold is applied twice). Every call to a filter-function calls `jpegStr` and `createFromJpegStr` once, which means that every filter-function converts the image into a format and then convert the image back to the original format. Accordingly, we should investigate if we can optmize this conversion. Finally, the second row shows that `load` is responsible by opening and reading each image.

By analyzing the application code B.1, we confirmed that every filter-function receives an image as a string, converts the image into a userdata by calling `createFromJpegStr`, modifies the userdata image, and then converts the userdata image back into a string image by calling `jpegStr`. We modified the original application and removed all conversions that the filter functions do. In the modified implementation B.2, instead of creating a new userdata and a new string on every call, each filter-function modifies the image in place. The modified implementation reduced the memory-usage measured by the time command from 2.14 gigabytes to 441 megabytes. It also reduced the number of mallocs by more than half and the malloc size by almost 5 times. Figure 4.8 shows the `luamemprofiler` final report for the modified version of the CAPTCHA JPEG Filter application.

4.3 Performance Analysis

In this section, we present the execution time and the memory overhead for profiling the six applications with `luamemprofiler` and `lmprof`. We exe-

```
                                call/tot  parents
index %mem      ret       self  call      name
                                call/tot  children
===============================================================
             220.6 MB  220.6 MB  10K/25K   threshold (...)
             152.9 MB  152.9 MB   5K/25K   grayscale (...)
             117.9 MB  117.9 MB   5K/25K   blur (...)
              45.2 MB   45.2 MB   5K/25K   invert (...)
[2]---78.93--536.6-MB--536.6-MB--25K-------jpegStr-[C]
===============================================================
             140.6 MB    0.0 MB   5K/5K    main chunk (...)
[5]---20.68--140.6-MB----0.0-MB---5K-------load-(...)
             140.4 MB  140.4 MB   5K/5K    read [C]
               0.3 MB    0.3 MB   5K/5K    open [C]
===============================================================
               0.5 MB    0.5 MB  10K/25K  threshold (...)
               0.2 MB    0.2 MB   5K/25K  grayscale (...)
               0.2 MB    0.2 MB   5K/25K  blur (...)
               0.2 MB    0.2 MB   5K/25K  invert (...)
[9]----0.17----1.1-MB----1.1-MB--25K------createFromJpegStr-[C]
```

Figure 4.7: `lmprof` call-graph report of the CAPTCHA JPEG Filter application. The format has been slightly altered to better fit this thesis layout.

cuted all tests in a notebook with two Intel® Core® Processors i7-2640M (4M Cache, 2.80GHz), for a total of 4 cores, 8GB RAM and 500GB SA-SCSI 7200 RPM hard drive. The notebook had Linux Mint 17 LTS Qiana (64 bit) installed with essential services running and the desktop interface loaded, which is needed for testing `luamemprofiler` display. The measures were done using the `time` Linux command. The execution time is the "elapsed real (wall clock) time used by the process" (%E option) and the memory usage is "the maximum resident set size of the process during its lifetime" (%M option).

We execute each application as *pure* Lua (no modifications to the application), *lmprof* (application profiled with `lmprof`), *lmp* (application profiled with `luamemprofiler` with the display turned off), and *lmpD* (application profiled with `luamemprofiler` with the display turned on). Also, for each application, we used inputs that generate light (L), medium (M), and heavy (H) loads. Except for SMM and PLC, light means an input load that executes in approximately 10 seconds. The medium input is 10 times larger and the heavy input is 100 times larger. We executed each test five times. To consolidate results, we removed one outlier from each test and calculated the means with the standard deviations.

Figure 4.9 shows the execution times. Regarding `lmprof`, the slowdown varies from 0x to 2.4x, which is much bigger than the `luamemprofiler`

```
Number of Mallocs  = 30673     Total Malloc Size  = 141.47 MB
Number of Reallocs =   170     Total Realloc Size =   0.13 MB
Number of Frees    = 25417     Total Free Size    = 141.25 MB

Maximum Memory Used = 1.29 MB

Number of Allocs of Each Type:
  String = 15412 | Function =  7 | Userdata = 15001
  Thread =     0 | Table    = 10 | Other    =   243
```

Figure 4.8: `luamemprofiler` final report of the "fixed" CAPTCHA JPEG Filter application. The format has been slightly altered to better fit this thesis layout.

slowdown. To understand the overhead imposed by `lmprof`, we compiled a new version of `lmprof` that counts the total number of functions calls during the program execution. Then, as Figure 4.10 shows, we executed each test with the heavy input and compared the number of function calls executed by the test to the application slowdown while being profiled with `lmprof`. As we can see, the number of function calls executed by a program has direct relation to `lmprof`'s overhead. While applications that do not have many function calls (e.g. BAS and SMM) suffer small slowdowns, applications that do many function calls (e.g. JPG and PLC) suffer big slowdowns. This happens because lmprof, in contrast to luamemprofiler, hooks every function call, instead of just the function calls that allocate data.

Regarding `luamemprofiler`, the overhead varies from 0% to 2% without display and from 0% to 13% with display, except by the PLC application. When the display is turned on, `luamemprofiler` slows down the PLC execution by 2.8x. PLC has a very large number of allocated and deleted blocks when compared to the other applications. It allocates approximately 24 million blocks, while other applications allocate 100 thousand. Also, differently from applications such as the BAS that allocates data at specific parts of the execution, PLC allocates and deallocates these blocks during the entire execution. Accordingly, drawing and erasing these blocks become expensive.

Figure 4.11 shows the memory overhead for each test. `lmprof` overhead is very low; it varies from 0% to 2% in most tests. Although PLC has many different function calls, which increases the `lmprof` overhead, we consider 8% a low overhead for a profiling tool. `luamemprofiler` also has low overhead in most profiled applications. However, as it records every allocated object, it is more expensive than `lmprof`, especially in programs that allocate many objects (e.g PLC and BAS). Comparing the real values of `lmp` and `lmpD`, the digital library has approximately 4 megabytes, which causes huge impact in

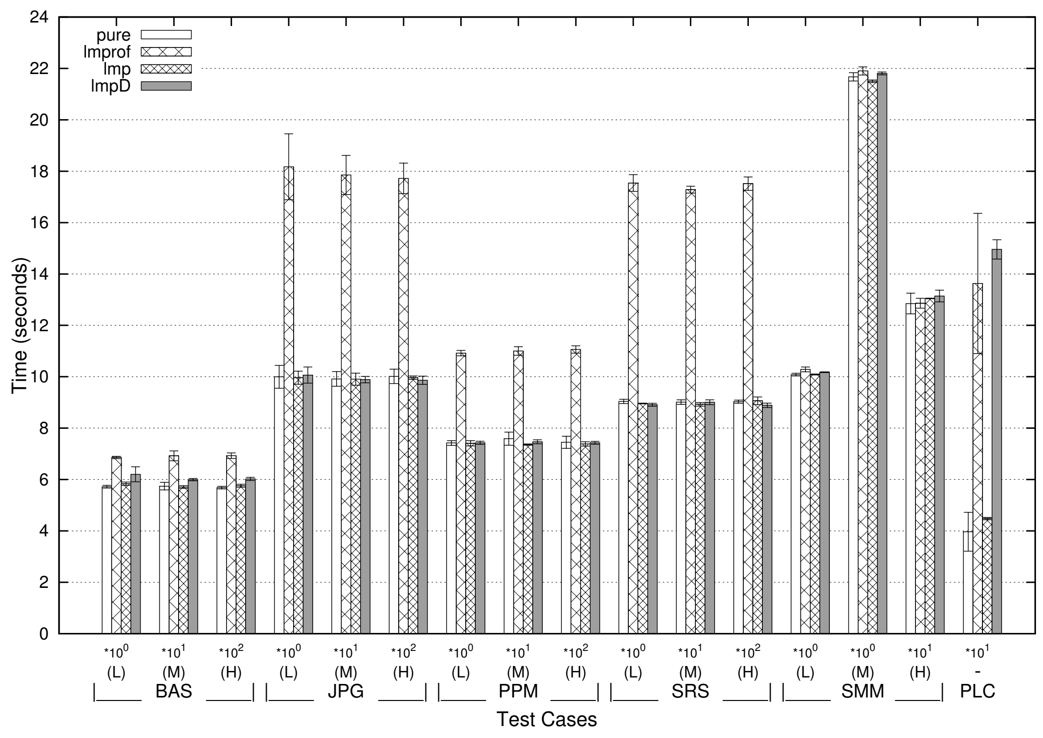


Figure 4.9: Execution time for each test in pure Lua, using `lmprof`, using `luamemprofiler` with the display turned off, and using `luamemprofiler` with the display turned on.

programs with a small memory footprint, such as PPM and SRS tests.

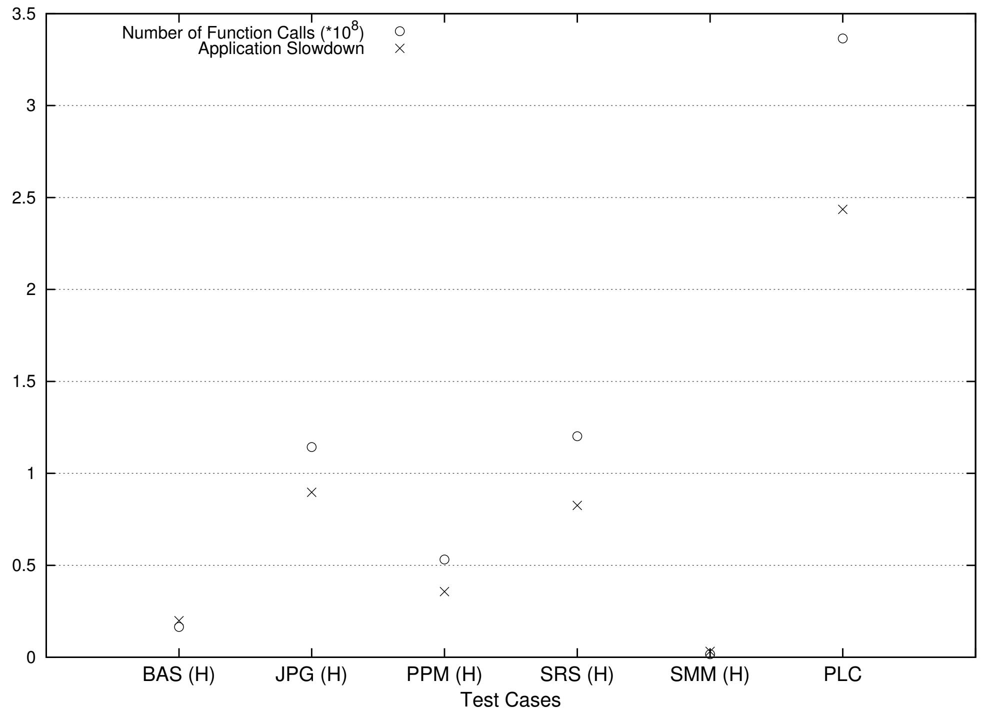


Figure 4.10: Relation between the number of function calls executed by each application and the slowdown imposed by `lmprof`.

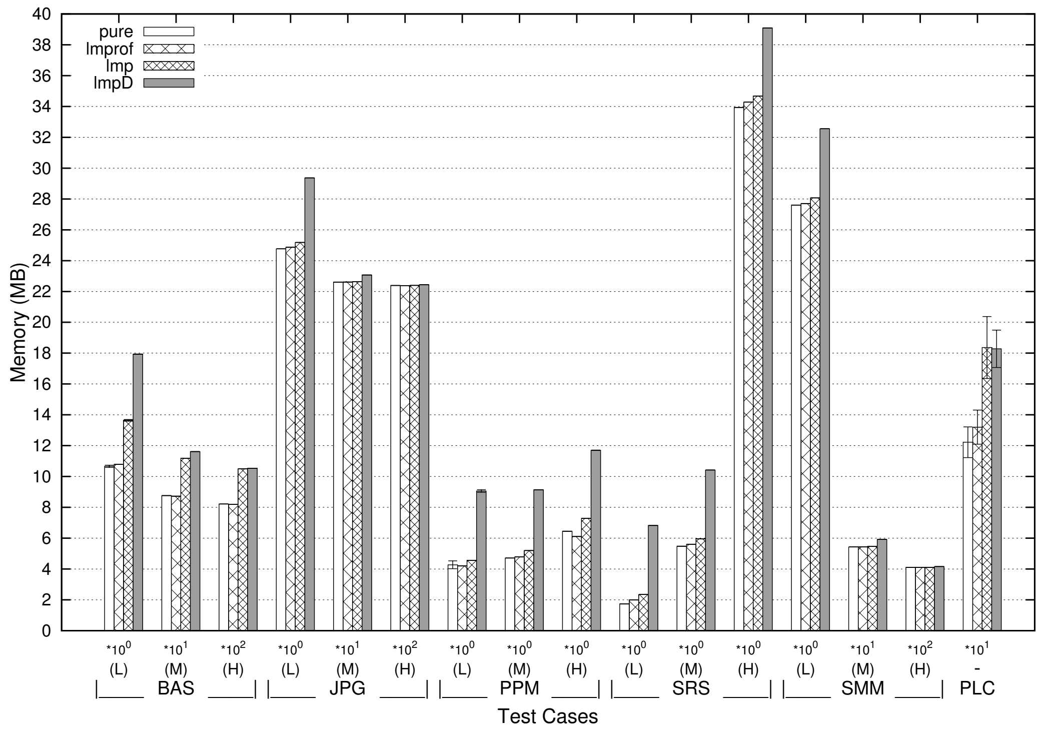


Figure 4.11: Memory consumption for each test in pure Lua, using `lmprof`, using `luamemprofiler` with the display turned off, and using `luamemprofiler` with the display turned on.

5 Conclusion

In this thesis, we presented a study on memory profilers and how they can help programmers identify and fix memory bloats. We analyzed published work and currently used tools in three scripting languages (JavaScript, Python, and Lua) and, based on how the programmer uses each tool, we divided memory profilers in two groups, manual and automatic.

`Heap Profiler` is the most advanced tool for scripting languages. It is a JavaScript tool built over Google Chrome that offers a manual and an automatic memory profiler with many visual features. Python has many manual tools that help programmers analyze specific objects. It also has a good automatic tool to analyze the overall behavior of a program. Lua has only three manual tools that analyze specific parts of the program. In summary, memory profilers in these languages explore few profiling techniques and research in this area should be encouraged.

Lua was really important to our research due to its simplicity and flexibility. The main goal of Lua is to be an embedded language, and for this reason it prefers to provide mechanisms instead of fixed policies to programmers. These mechanisms help developers easily instrument a program execution.

Lua's complete and fully reentrant API together with its easy integration with C are very handful to implement memory profilers. Developers can easily monitor and collect information about Lua programs from C, which allows memory-profiler developers to allocate all the auxiliary data apart from the memory used by the main program. As memory-profiler allocations do not interfere in the monitored memory, reports are easier to calculate and more precise.

The function `lua_sethook` exposes a powerful mechanism that allows developers to set up different hooks, such as a *return* event hook that collects the total amount of memory used by the program after each function returns. It is important to highlight that its implementation offers multiple hook events so that a developer can choose the events that best fit its use case. As an example, a *tail-call* event is different from a *call* event, so we used the flag that distinguishes them to handle tail calls correctly.

Another powerful mechanism is the ability to dynamically change the allocation function. The function `lua_setallocf`, available since 2006, allows developers to substitute the allocation function by a custom allocation function that, for example, collects metrics. Python has a similar mechanism implemented in version 3.4, which was released in 2014 [51, 46]. As a result of this implementation, in the same 3.4 release there was also a `tracemalloc` module to trace memory blocks allocated by Python [45]. We believe that one of the reasons for most Python's memory profilers being pretty printers is due to the lack of this mechanism in earlier versions. JavaScript does not expose such mechanism.

After studying different techniques, we implemented two automatic memory profilers: `luamemprofiler` and `lmprof`. We created `luamemprofiler` to explore real-time visualization, data categorization by type, and ongoing interaction. `lmprof` was created to explore function profiling.

Real-time visualization was useful to quickly understand the amount of memory that a program uses throughout execution (i.e. if blocks are allocated and garbage-collected shortly or if blocks are allocated and remain in the heap for a long time). The ongoing interaction and the block categorization by type were not as useful as we expected. Ongoing interaction was not useful because we implemented only simple interaction features, which proved to be hard to use in practice. For instance, the step-by-step execution is very hard to use in real programs if the programmer cannot set a breakpoint, which is the case. Accordingly, we believe that by adding more advanced features, the ongoing interaction will be helpful. Type categorization is very useful in many profilers and was somehow useful to analyze applications. However, due to the problems that we explained in Section 3.2.2, the categorization does not contain as much information as we wanted.

Finally, function profiling was very useful to understand unfamiliar programs. Usually, just by reading the top five functions the programmer knows the main execution flow that allocates data. It is important to highlight that we opted to monitor just function-calls that allocate data. Therefore, as many function calls do not allocate data, `lmprof` reports cannot be used to understand the complete execution flow. We do not have a final opinion about monitoring or not all the function calls. On the one hand, we would be able to understand the complete execution flow. On the other hand, too much information could obscure the memory information, which is the focus. Function profiling was also very useful to identify memory bloats. As the reports highlight the functions that allocated most of the memory, the programmer can easily narrow down the functions that should be analyzed regarding memory bloats.

We evaluated `luamemprofiler` and `lmprof` considering three important aspects. They should be easy to integrate into existing programs. Also, They should not impose too much overhead on the target program. And finally, they should provide readable reports for a regular programmer. We used six different applications developed by third party programmers to evaluate both tools.

Both `luamemprofiler` and `lmprof` are integrated into the target program by adding three lines to the target-program code. This is a very simple process that can be easily done manually. Also, the programmer can create a simple script to automatically add these lines, as we did to evaluate our tools. We could change the integration method to a simple parameter in the command line that executes the target program. However, by using this mechanism, our tools would always start profiling after being loaded, which can be a problem if a programmer wants to profile specific parts instead of the entire program. We need further investigation to decide which mechanism is better.

We evaluated overhead regarding both memory and execution time. The memory overhead imposed by our tools is usually low (less than 8%). In applications that allocate many objects, the overhead imposed by `luamemprofiler` increases (e.g. an application that allocated 24 million objects suffered 50% memory overhead). The execution time overhead imposed by `luamemprofiler` is usually low (less than 8%). However, in one of the tests the slowdown is 2.8x. The execution time overhead imposed by `lmprof` is on average 60%. The slowdown is relative to the number of function calls. In an application that did approximately 336 million function calls, the slowdown was 2.4x.

Finally, to evaluate the quality of the reports, we used our tools to analyze two applications that we were not familiar with. Just from the reports we were able to understand the overall behavior of the target program and pinpoint possible memory bloats. After analyzing and modifying the source code, we were able to reduce the memory usage of one application by approximately 4 times and of the other application by approximately 5 times.

Although our implementation focus on automatic memory profilers, we believe that manual memory profilers are also important. Moreover, we believe that integrating both types is very promising. We hope this thesis raises awareness regarding memory bloat, specially in scripting languages. Accordingly, we would like to highlight one future work for each tool. To `luamemprofiler`, we suggest the implementation of a debugger feature. By using that feature, the programmer will be able to set break points and execute commands inside the display. To `lmprof`, we suggest the implementation of a visual graph generator. Based on the metadata file, this script would create a visual graph of the function calls.

Bibliography

[1] BARRITZ, R.; COHEN, G. **Computer program profiler**, Feb. 11 2003. US Patent 6,519,766.

[2] BIENIA, C.; KUMAR, S.; SINGH, J. P. ; LI, K. **The PARSEC benchmark suite: Characterization and architectural implications**. In: proceedings of the 17th international conference on parallel architectures and compilation techniques, October 2008.

[3] BOND, M. D.; MCKINLEY, K. S. **Bell: Bit-encoding online memory leak detection**. In: proceedings of the 12th international conference on architectural support for programming languages and operating systems, ASPLOS XII, p. 61–72, New York, NY, USA, 2006. ACM.

[4] BOND, M. D.; MCKINLEY, K. S. **Tolerating memory leaks**. In: proceedings of the 23rd ACM SIGPLAN conference on object-oriented programming systems languages and applications, OOPSLA '08, p. 109–126, New York, NY, USA, 2008. ACM.

[5] BROUWERS, J.; HAEHNE, L. ; SCHUPPENIES, R. **Pympler**. `http://pythonhosted.org/Pympler/`, 2008. [Online; accessed 2015, Feb 18th].

[6] COPPA, E.; DEMETRESCU, C. ; FINOCCHI, I. **Input-sensitive profiling**. In: proceedings of the 33rd ACM SIGPLAN conference on programming language design and implementation, PLDI '12, p. 89–98, New York, NY, USA, 2012. ACM.

[7] DETLEFS, D. L.; KALSOW, B. **Debugging storage management problems in garbage-collected environments**. In: proceedings of the USENIX conference on object-oriented technologies on USENIX conference on object-oriented technologies, COOTS'95, p. 6, Berkeley, CA, USA, 1995. USENIX Association.

[8] FOWLER, M. Yet another optimization article. **IEEE Software**, v.19, n.3, p. 20–21, may 2002.

[9] GEDMINAS, M. **objgraph**. `http://mg.pov.lt/objgraph/`, 2010. [Online; accessed 2015, Feb 18th].

[10] GRAHAM, S. L.; KESSLER, P. B. ; MCKUSICK, M. K. **Gprof: A call graph execution profiler**. In: proceedings of the 1982 SIGPLAN symposium on compiler construction, SIGPLAN '82, p. 120–126, New York, NY, USA, 1982. ACM.

[11] HASTINGS, R.; JOYCE, B. **Purify: Fast detection of memory leaks and access errors**. In: in proceedings of the winter 1992 USENIX conference, p. 125–138, 1991.

[12] IERUSALIMSCHY, R. **Programming in Lua**. 3rd. ed., Lua.Org, 2013.

[13] JANDA, P. **microscope — visualizing complex lua values using graphviz**. `http://siffiejoe.github.io/lua-microscope/`, 2013. [Online; accessed 2015, Feb 18th].

[14] JUMP, M.; MCKINLEY, K. S. **Cork: Dynamic memory leak detection for garbage-collected languages**. In: proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL '07, p. 31–38, New York, NY, USA, 2007. ACM.

[15] KNUTH, D. E. The art of computer programming, volume 1: Fundamental algorithms. **Addison-Wesley, Reading, Mass**, 1973.

[16] MITCHELL, N.; SCHONBERG, E. ; SEVITSKY, G. Four trends leading to Java runtime bloat. **IEEE Software**, v.27, n.1, p. 56–63, Jan. 2010.

[17] MITCHELL, N.; SEVITSKY, G. **LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications**. In: proceedings of the 17th european conference on object-oriented programming, volume 2743 of **Lecture Notes in Computer Science**, p. 351–377. Springer Berlin Heidelberg, 2003.

[18] NETHERCOTE, N.; SEWARD, J. **Valgrind: A framework for heavyweight dynamic binary instrumentation**. In: proceedings of the 2007 ACM SIGPLAN conference on programming language design and implementation, PLDI '07, p. 89–100, New York, NY, USA, 2007. ACM.

[19] NILSSON, S. **Guppy Heapy**. `http://guppy-pe.sourceforge.net/`, 2006. [Online; accessed 2015, Feb 18th].

[20] PAUW, W. D.; SEVITSKY, G. **Visualizing reference patterns for solving memory leaks in Java**. In: proceedings of the 13th european conference on object-oriented programming, ECOOP '99, p. 116–134, London, UK, UK, 1999. Springer-Verlag.

[21] PEDREGOSA, F.; GERVAIS, P. **Memory Profiler**. `http://pypi.python.org/pypi/memory_profiler`, 2011. [Online; accessed 2015, Feb 18th].

[22] PERKIN, L. **ProFi**. `https://gist.github.com/perky/2838755`, 2012. [Online; accessed 2015, Feb 18th].

[23] PIENAAR, J. A.; HUNDT, R. **JSWhiz: Static analysis for JavaScript memory leaks**. In: proceedings of the 2013 IEEE/ACM international symposium on code generation and optimization, CGO '13, p. 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[24] ROGERS, J. **Understanding and solving Internet Explorer leak patterns**. `https://goo.gl/CI3pwu`, 2005. [Online; accessed 2014, April 1st].

[25] ROZENDORN, G. **dowser**. `https://github.com/Infinidat/dowser`, 2014. [Online; accessed 2015, Feb 18th].

[26] SERRANO, M.; BOEHM, H.-J. **Understanding memory allocation of Scheme programs**. In: proceedings of the fifth ACM SIGPLAN international conference on functional programming, ICFP '00, p. 245–256, New York, NY, USA, 2000. ACM.

[27] SMITH, L. A.; BULL, J. M. ; OBDRZÁLEK, J. **A parallel Java Grande benchmark suite**. In: proceedings of the 2001 ACM/IEEE conference on supercomputing, SC '01, p. 8–8, New York, NY, USA, 2001. ACM.

[28] ŠOR, V.; SRIRAMA, S. N. Memory leak detection in Java: Taxonomy and classification of approaches. **Journal of Systems and Software**, v.96, p. 139–151, 2014.

[29] SUN, J.; GEHRINGER, E. F. **A Smalltalk memory profiler and its performance enhancement**. In: presented at OOPSLA '97 memory managemnt and garbage collection workshop, 1997.

[30] TANG, Y.; TANG, Y.; GAO, Q.; GAO, Q.; QIN, F. ; QIN, F. **LeakSurvivor: Towards safely tolerating memory leaks for garbage-collected languages**. In: USENIX 2008 annual technical conference, ATC'08, p. 307–320, Berkeley, CA, USA, 2008. USENIX Association.

[31] WILD, M. **luatraverse**. `http://code.matthewwild.co.uk/luatraverse/`, 2006. [Online; accessed 2015, Feb 18th].

[32] XU, G.; MITCHELL, N.; ARNOLD, M.; ROUNTEV, A.; SCHONBERG, E. ; SEVITSKY, G. Scalable runtime bloat detection using abstract dynamic slicing. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, v.23, n.3, p. 23, 2014.

[33] ZORN, B.; HILFINGER, P. A memory allocation profiler for C and Lisp programs. **Proceedings of the Summer USENIX**, p. 1–15, 1988.

[34] **Find all references to an object in Lua**. `http://stackoverflow.com/questions/7166124`. [Online; accessed 2015, April 16th].

[35] **Help with memory leak**. `http://giderosmobile.com/forum/discussion/2645`. [Online; accessed 2015, May 17th].

[36] **Lua table memory leak?** `http://stackoverflow.com/questions/20139359`. [Online; accessed 2015, April 16th].

[37] **Simple DirectMedia Layer**. `https://www.libsdl.org/index.php`, 1998. [Online; accessed 2015, May 17th].

[38] **Google summer of code**. `https://code.google.com/p/google-summer-of-code/`, 2004. [Online; accessed 2014, Oct 2nd].

[39] **Python memory leaks**. `http://stackoverflow.com/questions/1435415`, 2009. [Online; accessed 2015, May 2nd].

[40] **Lua memory profiler**. `http://www.a1k0n.net/code/lallocprof/`, 2011. [Online; accessed 2015, Feb 18th].

[41] **Memory leak protection**. `http://wiki.apache.org/tomcat/MemoryLeakProtection`, 2012. [Online; accessed 2014, April 1st].

[42] **Profiling**. `http://en.wikipedia.org/wiki/Profiling_(computer_programming)`, 2012. [Online; accessed 2015, May 17th].

[43] **Reducing memory usage - Firefox**. `http://kb.mozillazine.org/Memory_Leak#Memory_leaks`, 2013. [Online; accessed 2014, April 1st].

[44] **TIOBE index**. `http://www.tiobe.com/`, 2014. [Online; accessed 2014, March 15th].

[45] **Add a new tracemalloc module to trace Python memory allocations**. `https://www.python.org/dev/peps/pep-0454/`, 2015. [Online; accessed 2015, May 29th].

[46] **Add new APIs to customize Python memory allocators.** `https://www.python.org/dev/peps/pep-0445/`, 2015. [Online; accessed 2015, May 29th].

[47] **Batteries included.** `https://docs.python.org/2/tutorial/stdlib.html#batteries-included`, 2015. [Online; accessed 2015, May 29th].

[48] **Chrome DevTools.** `https://developer.chrome.com/devtools`, 2015. [Online; accessed 2015, Feb 18th].

[49] **Javascript closures and memory leaks with a lot of imbricated functions and callbacks.** `http://stackoverflow.com/questions/29926623`, 2015. [Online; accessed 2015, May 2nd].

[50] **JavaScript typed arrays.** `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays`, 2015. [Online; accessed 2015, May 14th].

[51] **What's new in python 3.4.** `https://docs.python.org/3.4/whatsnew/3.4.html`, 2015. [Online; accessed 2015, May 29th].

A
Black and Scholes Source Code

A.1
Original Application

```
--
-- Black-Scholes
-- Analytical method for calculating European Options
--
-- Ported from blackscholes.c included in the PARSEC benchmark suite.
--

if ( #arg < 2 ) then
  print( "usage: " .. arg[0] .. " <input_file> <output_file>" )
  return
end

-- number of runs (hard-coded in original implementation)
numruns = 100

-- assign command line parameters
nthreads = 1
infile  = arg[1]
outfile = arg[2]

-- open input file
infh = io.open( infile )
if ( infh == nil ) then
  print( "cannot open input file " .. infile )
  return
end

-- read number of options in first line of input file
```

```
numoptions = tonumber(infh:read( "n" ))
if ( numoptions == nil ) then
  print( "cannot read number of options from input file " .. infile )
  return
end

-- adjust number of threads, if necessary, to suit number of options
if ( nthreads > numoptions ) then
  print( "not enough work to keep all threads busy; " ..
         "reducing number of threads." )
  nthreads = numoptions
end

-- advance file pointer to second line
infh:read( "l" )

-- read options data from input file, starting at line 2
datatb = {}
fnametb = { "s", "strike", "r", "divq", "v", "t", "optiontype", "divs",
            "dgrefval" }
for line in infh:lines() do
  optiontb = {}
  fnum = 1
  for field in string.gmatch( line, "[^%s]+" ) do
    --table.insert( optiontb, field )
    optiontb[fnametb[fnum]] = field
    fnum = fnum + 1
  end
  table.insert( datatb, optiontb )
end

-- close input file
infh:close()

-- print initial information
print("Num of Options: " .. numoptions )
print("Num of Runs: " .. numruns )

-- divide option fields into records
```

```
otype = {}
sptprice = {}
strike = {}
rate = {}
volatility = {}
otime = {}
for i = 1, numoptions, 1 do
  if ( datatb[i].optiontype == 'P' ) then
    table.insert( otype, 1 )
  else
    table.insert( otype, 0 )
  end
  table.insert( sptprice, datatb[i].s )
  table.insert( strike, datatb[i].strike )
  table.insert( rate, datatb[i].r )
  table.insert( volatility, datatb[i].v )
  table.insert( otime, datatb[i].t )
end

-- Cumulative Normal Distribution Function
function CNDF( InputX )

  inv_sqrt_2xPI = 0.39894228040143270286

  if ( InputX < 0 ) then
    InputX = -InputX
    sign = 1
  else
    sign = 0
  end

  xInput = InputX

  -- Compute NPrimeX term common to both four & six decimal accuracy calcs
  expValues = math.exp( -0.5 * InputX * InputX )
  xNPrimeofX = expValues
  xNPrimeofX = xNPrimeofX * inv_sqrt_2xPI

  xK2 = 0.2316419 * xInput
  xK2 = 1.0 + xK2
  xK2 = 1.0 / xK2
  xK2_2 = xK2 * xK2
  xK2_3 = xK2_2 * xK2
  xK2_4 = xK2_3 * xK2
  xK2_5 = xK2_4 * xK2

  xLocal_1 = xK2 * 0.319381530
  xLocal_2 = xK2_2 * (-0.356563782)
  xLocal_3 = xK2_3 * 1.781477937
  xLocal_2 = xLocal_2 + xLocal_3
  xLocal_3 = xK2_4 * (-1.821255978)
  xLocal_2 = xLocal_2 + xLocal_3
  xLocal_3 = xK2_5 * 1.330274429
  xLocal_2 = xLocal_2 + xLocal_3

  xLocal_1 = xLocal_2 + xLocal_1
  xLocal   = xLocal_1 * xNPrimeofX
  xLocal   = 1.0 - xLocal

  OutputX  = xLocal

  if ( sign ~= 0 ) then
      OutputX = 1.0 - OutputX;
  end

  return OutputX

end

-- main Black & Schole's equation function
function BlkSchlsEqEuroNoDiv( sptprice, strike, rate, volatility, time,
                              otype, timet )

  xStockPrice = sptprice
  xStrikePrice = strike
  xRiskFreeRate = rate
  xVolatility = volatility

  xTime = time;
  xSqrtTime = math.sqrt( xTime )

  logValues = math.log( sptprice / strike )

  xLogTerm = logValues

  xPowerTerm = xVolatility * xVolatility
  xPowerTerm = xPowerTerm * 0.5

  xD1 = xRiskFreeRate + xPowerTerm
  xD1 = xD1 * xTime
  xD1 = xD1 + xLogTerm

  xDen = xVolatility * xSqrtTime
  xD1 = xD1 / xDen
  xD2 = xD1 -  xDen

  d1 = xD1
  d2 = xD2

  NofXd1 = CNDF( d1 )
  NofXd2 = CNDF( d2 )

  FutureValueX = strike * (math.exp( -( rate )*( time )))
  if ( otype == 0 ) then
    OptionPrice = (sptprice * NofXd1) - (FutureValueX * NofXd2)
  else
    NegNofXd1 = (1.0 - NofXd1)
    NegNofXd2 = (1.0 - NofXd2)
    OptionPrice = (FutureValueX * NegNofXd2) - (sptprice * NegNofXd1)
  end

  return OptionPrice

end

-- worker thread
function bs_thread( tid )
```

```
  from = (tid - 1) * (numoptions / nthreads ) + 1
  to   = from + (numoptions / nthreads ) - 1
  for j = 1, numruns, 1 do
    for i = from, to, 1 do
      -- call main function to calculate option value based on
      -- Black & Schole's equation
      p = BlkSchlsEqEuroNoDiv( sptprice[i], strike[i], rate[i],
                               volatility[i], otime[i], otype[i], 0 )
      pricestb[i] = p
    end
  end
end

-- create results table
pricestb = {}

-- create threads
for i = 1, nthreads, 1 do
  co = coroutine.create( bs_thread )
  coroutine.resume( co, i )
end

-- write results to outfile
outfh = io.open( outfile, "w" )
if ( outfh == nil ) then
  print( "cannot open output file " .. outfile )
  return
end

outfh:write( numoptions, "\n" )
for i = 1, numoptions, 1 do
  outfh:write( string.format( "%.18f", pricestb[i] ), "\n" )
end

outfh:close()
```

A.2 Modified Application

```
--
```

```lua
-- Black-Scholes
-- Analytical method for calculating European Options
--
-- Ported from blackscholes.c included in the PARSEC benchmark suite.
--

if ( #arg < 2 ) then
  print( "usage: " .. arg[0] .. " <input_file> <output_file>" )
  return
end

-- number of runs (hard-coded in original implementation)
numruns = 100

-- assign command line parameters
nthreads = 1
infile  = arg[1]
outfile = arg[2]

-- open input file
infh = io.open( infile )
if ( infh == nil ) then
  print( "cannot open input file " .. infile )
  return
end

-- read number of options in first line of input file
numoptions = tonumber(infh:read( "n" ))
if ( numoptions == nil ) then
  print( "cannot read number of options from input file " .. infile )
  return
end

-- adjust number of threads, if necessary, to suit number of options
if ( nthreads > numoptions ) then
  print( "not enough work to keep all threads busy; " ..
         "reducing number of threads." )
  nthreads = numoptions
end
```

```

-- advance file pointer to second line
infh:read( "l" )

-- read options data from input file, starting at line 2
--datatb = {}
fnametb = { "s", "strike", "r", "divq", "v", "t", "optiontype", "divs",
            "dgrefval" }

-- divide option fields into records
otype = {}
sptprice = {}
strike = {}
rate = {}
volatility = {}
otime = {}

for line in infh:lines() do
  optiontb = {}
  fnum = 1
  for field in string.gmatch( line, "[^%s]+" ) do
    --table.insert( optiontb, field )
    optiontb[fnametb[fnum]] = field
    fnum = fnum + 1
  end

  if ( optiontb.optiontype == 'P' ) then
    table.insert( otype, 1 )
  else
    table.insert( otype, 0 )
  end
  table.insert( sptprice, optiontb.s )
  table.insert( strike, optiontb.strike )
  table.insert( rate, optiontb.r )
  table.insert( volatility, optiontb.v )
  table.insert( otime, optiontb.t )
end

-- close input file
```

```
infh:close()

-- print initial information
print("Num of Options: " .. numoptions )
print("Num of Runs: " .. numruns )

-- Cumulative Normal Distribution Function
function CNDF( InputX )

  inv_sqrt_2xPI = 0.39894228040143270286

  if ( InputX < 0 ) then
    InputX = -InputX
    sign = 1
  else
    sign = 0
  end

  xInput = InputX

  -- Compute NPrimeX term common to both four & six decimal accuracy calcs
  expValues = math.exp( -0.5 * InputX * InputX )
  xNPrimeofX = expValues
  xNPrimeofX = xNPrimeofX * inv_sqrt_2xPI

  xK2 = 0.2316419 * xInput
  xK2 = 1.0 + xK2
  xK2 = 1.0 / xK2
  xK2_2 = xK2 * xK2
  xK2_3 = xK2_2 * xK2
  xK2_4 = xK2_3 * xK2
  xK2_5 = xK2_4 * xK2

  xLocal_1 = xK2 * 0.319381530
  xLocal_2 = xK2_2 * (-0.356563782)
  xLocal_3 = xK2_3 * 1.781477937
  xLocal_2 = xLocal_2 + xLocal_3
  xLocal_3 = xK2_4 * (-1.821255978)
  xLocal_2 = xLocal_2 + xLocal_3
  xLocal_3 = xK2_5 * 1.330274429
  xLocal_2 = xLocal_2 + xLocal_3

  xLocal_1 = xLocal_2 + xLocal_1
  xLocal   = xLocal_1 * xNPrimeofX
  xLocal   = 1.0 - xLocal

  OutputX  = xLocal

  if ( sign ~= 0 ) then
      OutputX = 1.0 - OutputX;
  end

  return OutputX

end

-- main Black & Schole's equation function
function BlkSchlsEqEuroNoDiv( sptprice, strike, rate, volatility, time,
                              otype, timet )

  xStockPrice = sptprice
  xStrikePrice = strike
  xRiskFreeRate = rate
  xVolatility = volatility

  xTime = time;
  xSqrtTime = math.sqrt( xTime )

  logValues = math.log( sptprice / strike )

  xLogTerm = logValues

  xPowerTerm = xVolatility * xVolatility
  xPowerTerm = xPowerTerm * 0.5

  xD1 = xRiskFreeRate + xPowerTerm
  xD1 = xD1 * xTime
  xD1 = xD1 + xLogTerm

  xDen = xVolatility * xSqrtTime
  xD1 = xD1 / xDen
  xD2 = xD1 -  xDen

  d1 = xD1
  d2 = xD2

  NofXd1 = CNDF( d1 )
  NofXd2 = CNDF( d2 )

  FutureValueX = strike * (math.exp( -( rate )*( time )))
  if ( otype == 0 ) then
    OptionPrice = (sptprice * NofXd1) - (FutureValueX * NofXd2)
  else
    NegNofXd1 = (1.0 - NofXd1)
    NegNofXd2 = (1.0 - NofXd2)
    OptionPrice = (FutureValueX * NegNofXd2) - (sptprice * NegNofXd1)
  end

  return OptionPrice

end

-- worker thread
function bs_thread( tid )
  from = (tid - 1) * (numoptions / nthreads ) + 1
  to   = from + (numoptions / nthreads ) - 1
  for j = 1, numruns, 1 do
    for i = from, to, 1 do
      -- call main function to calculate option value based on
      -- Black & Schole's equation
      p = BlkSchlsEqEuroNoDiv( sptprice[i], strike[i], rate[i],
                               volatility[i], otime[i], otype[i], 0 )
      pricestb[i] = p
    end
  end
end

```

```
-- create results table
pricestb = {}

-- create threads
for i = 1, nthreads, 1 do
  co = coroutine.create( bs_thread )
  coroutine.resume( co, i )
end

-- write results to outfile
outfh = io.open( outfile, "w" )
if ( outfh == nil ) then
  print( "cannot open output file " .. outfile )
  return
end

outfh:write( numoptions, "\n" )
for i = 1, numoptions, 1 do
  outfh:write( string.format( "%.18f", pricestb[i] ), "\n" )
end

outfh:close()
```

B
CAPTCHA JPEG Filter Source Code

B.1
Original Application

B.1.1
main.lua

```lua
--
-- CAPTCHA JPEG Filter
-- an application to filter CAPTCHA images in the JPEG format to make it
-- easier to perform automatic optical character recognition (OCR).
--

local image = require"image"
local lfs = require"lfs"

if ( #arg < 2 ) then
  print( "usage: " .. arg[0] .. " <input_image_dir> <output_image_dir>" )
  return
end

indir    = arg[1]
outdir   = arg[2]
startdir = lfs.currentdir()

if ( not lfs.chdir( indir )) then
  io.stderr:write( "cannot change to input dir \"" .. indir .. "\"\n" )
  return
end

local imgfiles = {}
for f in lfs.dir( lfs.currentdir( )) do
  if ( lfs.attributes( f, "mode" ) == "file" ) then
```

```
    if ( string.match( f, "%.jpg$" ) or
         string.match( f, "%.JPG$" )) then
      table.insert( imgfiles, f )
    end
  end
end

lfs.chdir( startdir )

for _,f in pairs( imgfiles ) do
  local img, err = image.load( indir .. "/" .. f )
  if ( not img ) then
    io.stderr:write( err .. "\n" )
    return
  end
  img = image.grayscale( img )
  img = image.threshold( img, 220 )
  img = image.blur( img, 1 )
  img = image.threshold( img, 70 )
  img = image.invert( img )
  image.save( img, outdir .. "/" .. f )
end
```

B.1.2
image.lua

```
local math = require"math"
local gd = require"gd"
local io = require"io"

-- fix for "FILE* expected, got FILE*" bug in io library
if ( io ) then
  getmetatable(io.input()).__gc = nil
end

image = {}

image.load =
  function( infile )
    -- open input file
    local infh = io.open( infile, "rb" )
    if ( infh == nil ) then
      return false, "cannot open input file " .. infile
    end
    -- read the whole file
    local data = infh:read( "a" )
    -- close file
    infh:close()
    -- return data read from file
    return data
  end

image.grayscale =
  function( img )
    local gdimg = gd.createFromJpegStr( img )
    getmetatable(gdimg).__gc = nil
    for i = 0, gdimg:sizeX()-1, 1 do
      for j = 0, gdimg:sizeY()-1, 1 do
        local r = gdimg:red(gdimg:getPixel(i,j))
        local g = gdimg:green(gdimg:getPixel(i,j))
        local b = gdimg:blue(gdimg:getPixel(i,j))
        local avg = math.modf((r * 0.3) + (g * 0.59) + (b * 0.11))
        gdimg:setPixel(i,j,gdimg:colorExact(avg,avg,avg))
      end
    end
  return gdimg:jpegStr(100)
  end

image.threshold =
  function( img, thresh )
    local gdimg = gd.createFromJpegStr( img )
    getmetatable(gdimg).__gc = nil
    for i = 0, gdimg:sizeX()-1, 1 do
      for j = 0, gdimg:sizeY()-1, 1 do
        -- since img is grayscale, we can get r,g,b (they're all equal)
        local gray = gdimg:red(gdimg:getPixel(i,j))
        if ( gray < thresh ) then
--          gdimg:setPixel(i,j,gdimg:colorExact(max,max,max))
          gdimg:setPixel(i,j,gdimg:colorExact(0,0,0))
        else
--          gdimg:setPixel(i,j,gdimg:colorExact(0,0,0))
          gdimg:setPixel(i,j,gdimg:colorExact(255,255,255))
        end
      end
    end
  return gdimg:jpegStr(100)
  end

image.blur =
  function( img, blursize )
    local gdimg = gd.createFromJpegStr( img )
    getmetatable(gdimg).__gc = nil
    local w = gdimg:sizeX()
    local h = gdimg:sizeY()
    for i = 0, w-1, 1 do
      for j = 0, h-1, 1 do
        local avgr = 0
        local avgg = 0
        local avgb = 0
        local blurpxcount = 0
        for x = i, i + blursize, 1  do
          if x >= w then break end
          for y = j, j + blursize, 1 do
            if y >= h then break end
            avgr = avgr + gdimg:red(gdimg:getPixel(x,y))
            avgg = avgg + gdimg:green(gdimg:getPixel(x,y))
            avgb = avgb + gdimg:blue(gdimg:getPixel(x,y))
            blurpxcount = blurpxcount + 1
          end
        end
        avgr = math.floor( avgr / blurpxcount )
        avgg = math.floor( avgg / blurpxcount )
        avgb = math.floor( avgb / blurpxcount )
        for x = i, i + blursize, 1  do
          if x >= w then break end
          for y = j, j + blursize, 1 do
            if y >= h then break end
            -- gd max for alpha channel is 127
          gdimg:setPixel(x,y,gdimg:colorExact(avgr,avgg,avgb))
        end
      end
    end
  end
return gdimg:jpegStr(100)
end

image.invert =
  function( img )
    local gdimg = gd.createFromJpegStr( img )
    getmetatable(gdimg).__gc = nil
    for i = 0, gdimg:sizeX()-1, 1 do
      for j = 0, gdimg:sizeY()-1, 1 do
        -- since img is grayscale, we can get r,g,b (they're all equal)
        local gray = gdimg:red(gdimg:getPixel(i,j))
        local inv  = 255 - gray
        gdimg:setPixel(i,j,gdimg:colorExact(inv,inv,inv))
      end
    end
  return gdimg:jpegStr(100)
  end

image.save =
  function( img, outfile )
    -- open output file
    local outfh = io.open( outfile, "wb" )
    if ( outfh == nil ) then
      return false, "cannot open output file " .. outfile
    end
    local f, err = outfh:write( img )
    outfh:flush()
    -- close file
    outfh:close()
    -- return
    if ( not f ) then
      return false, "error writing to outputfile: " .. err
    end
    return true
  end

return image
```

B.2 Modified Application

B.2.1 main.lua

```
--
-- CAPTCHA JPEG Filter
-- an application to filter CAPTCHA images in the JPEG format to make it
-- easier to perform automatic optical character recognition (OCR).
--

local image = require"image"
local lfs = require"lfs"

if ( #arg < 2 ) then
  print( "usage: " .. arg[0] .. " <input_image_dir> <output_image_dir>" )
  return
end

indir    = arg[1]
outdir   = arg[2]
startdir = lfs.currentdir()

if ( not lfs.chdir( indir )) then
  io.stderr:write( "cannot change to input dir \"" .. indir .. "\"\n" )
  return
end

local imgfiles = {}
for f in lfs.dir( lfs.currentdir( )) do
  if ( lfs.attributes( f, "mode" ) == "file" ) then
    if ( string.match( f, "%.jpg$" ) or
         string.match( f, "%.JPG$" )) then
      table.insert( imgfiles, f )
    end
  end
end

lfs.chdir( startdir )

for _,f in pairs( imgfiles ) do
  local img, err = image.load( indir .. "/" .. f )
  if ( not img ) then
    io.stderr:write( err .. "\n" )
    return
  end
  image.modify(img)
  image.grayscale( img )
  image.threshold( img, 220 )
  image.blur( img, 1 )
  image.threshold( img, 70 )
  image.invert( img )
  image.save( img, outdir .. "/" .. f )
end
```

B.2.2 image.lua

```
local math = require"math"
local gd = require"gd"
local io = require"io"

-- fix for "FILE* expected, got FILE*" bug in io library
if ( io ) then
  getmetatable(io.input()).__gc = nil
end

image = {}

image.load =
  function( infile )
    -- open input file
    local infh = io.open( infile, "rb" )
    if ( infh == nil ) then
      return false, "cannot open input file " .. infile
    end
    -- read the whole file
    local data = infh:read( "a" )
    -- close file
    infh:close()
    local gdimg = gd.createFromJpegStr(data)
    -- return data read from file
    return gdimg
  end

image.grayscale =
  function( gdimg )
    getmetatable(gdimg).__gc = nil
    for i = 0, gdimg:sizeX()-1, 1 do
      for j = 0, gdimg:sizeY()-1, 1 do
        local r = gdimg:red(gdimg:getPixel(i,j))
        local g = gdimg:green(gdimg:getPixel(i,j))
        local b = gdimg:blue(gdimg:getPixel(i,j))
        local avg = math.modf((r * 0.3) + (g * 0.59) + (b * 0.11))
        gdimg:setPixel(i,j,gdimg:colorExact(avg,avg,avg))
      end
    end
  end

image.threshold =
  function( gdimg, thresh )
    getmetatable(gdimg).__gc = nil
    for i = 0, gdimg:sizeX()-1, 1 do
      for j = 0, gdimg:sizeY()-1, 1 do
        -- since img is grayscale, we can get r,g,b (they're all equal)
        local gray = gdimg:red(gdimg:getPixel(i,j))
        if ( gray < thresh ) then
--          gdimg:setPixel(i,j,gdimg:colorExact(max,max,max))
          gdimg:setPixel(i,j,gdimg:colorExact(0,0,0))
        else
--          gdimg:setPixel(i,j,gdimg:colorExact(0,0,0))
          gdimg:setPixel(i,j,gdimg:colorExact(255,255,255))
        end
      end
    end
  end

image.blur =
  function( gdimg, blursize )
    getmetatable(gdimg).__gc = nil
    local w = gdimg:sizeX()
    local h = gdimg:sizeY()
    for i = 0, w-1, 1 do
      for j = 0, h-1, 1 do
        local avgr = 0
        local avgg = 0
        local avgb = 0
        local blurpxcount = 0
        for x = i, i + blursize, 1  do
          if x >= w then break end
          for y = j, j + blursize, 1 do
            if y >= h then break end
            avgr = avgr + gdimg:red(gdimg:getPixel(x,y))
            avgg = avgg + gdimg:green(gdimg:getPixel(x,y))
            avgb = avgb + gdimg:blue(gdimg:getPixel(x,y))
            blurpxcount = blurpxcount + 1
          end
        end
        avgr = math.floor( avgr / blurpxcount )
        avgg = math.floor( avgg / blurpxcount )
        avgb = math.floor( avgb / blurpxcount )
        for x = i, i + blursize, 1  do
          if x >= w then break end
          for y = j, j + blursize, 1 do
            if y >= h then break end
            -- gd max for alpha channel is 127
            gdimg:setPixel(x,y,gdimg:colorExact(avgr,avgg,avgb))
          end
        end
      end
    end
  end

```

```
image.invert =
  function( gdimg )
    getmetatable(gdimg).__gc = nil
    for i = 0, gdimg:sizeX()-1, 1 do
      for j = 0, gdimg:sizeY()-1, 1 do
        -- since img is grayscale, we can get r,g,b (they're all equal)
        local gray = gdimg:red(gdimg:getPixel(i,j))
        local inv  = 255 - gray
        gdimg:setPixel(i,j,gdimg:colorExact(inv,inv,inv))
      end
    end
  end

image.save =
  function( gdimg, outfile )
    local img = gd.jpegStr(gdimg, 100)
    -- open output file
    local outfh = io.open( outfile, "wb" )
    if ( outfh == nil ) then
      return false, "cannot open output file " .. outfile
    end
    local f, err = outfh:write( img )
    outfh:flush()
    -- close file
    outfh:close()
    -- return
    if ( not f ) then
      return false, "error writing to outputfile: " .. err
    end
    return true
  end

return image
```