STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIX**, Number 2, 2014

AN APPROACH TO RESOURCE MANAGEMENT OF C/C++ APPLICATIONS

HANNELORE MĂRGINEAN AND SIMONA MOTOGNA

Abstract. In this paper we will present an approach to resource management of C/C++ applications, that has been concluded by building a new and unique profiling tool, remgrind. We first introduce general notions about profiling and performance analysis and continue with describing Valgrind, the framework that we used to build our tool. Finally we provide details about our implementation and compare it to similar tools.

1. Introduction

Software complexity has increased significantly over the last few decades and finding errors or performance issues is an increasingly difficult task. A way to alleviate these issues are tools that help finding bugs or performance hotspots in the software by analyzing the program. Program analysis is a method used to examine the correctness and performance of software, in order to solve possible flaws and optimize the software to execute faster and use fewer resources.

There exists a large number of research contributions and tools that perform such an analysis at the level of basic blocks (Pixie [2], QPT [9]) or tracing tools (WRL Titan [1], MPTRACE [6]). With the appearance of modular programming paradigm, and the significant increase of software complexity, several issues have arisen, such that basic programming analysis tools do not provide sufficient information for development and maintenance teams. Nowadays, software applications are usually developed by large teams, and may use third party components or libraries that do not necessarily exhibit implementation details. As a consequence, if someone would want to evaluate the performance of his own code, he must take into consideration aspects regarding functions calls and memory used by these calls.

Received by the editors: October 19,2014.

2010 *Mathematics Subject Classification.* 68M20.

1998 *CR Categories and Descriptors.* H.3.4 [**Information Storage and Retrieval**]: Systems and Software – *Performance evaluation (efficiency and effectiveness)*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs – *Functional constructs*.

Key words and phrases. program analysis, profiling, resource management.

Our approach analyzes the resources used by an application at the function level in order to be able to find performance hotspots. Our tool *remgrind* monitors memory allocations and deallocations as well as I/O operations for C/C++ applications and annotates each function with this information. It is our hope that this unique approach can be used by developers to channel their optimization efforts to parts of the program that consume the most resources. *Remgrind* was not designed to obtain results that other tools already provide and as such is not a complete solution to profiling and testing applications, but it offers its unique view into programs. Similar profilers account for the running time of called routines in the running time of the routines that call them [8], or discover hidden asymptotic inefficiencies in the code [4]. Similar tools are presented in section 4.

The rest of the paper is structured as follows: Section 2 introduces the basic concepts of profiling, section 3 introduces our tool *remgrind* and explain it's feature and algorithms used in the approach, together with a case study. Related work section evaluates our tool against similar approaches and discusses it's specific features that differentiate it. In the end, we draw some conclusions and suggest some future extensions.

2. Profiling

Program analysis can be classified in two major approaches: static and dynamic.

Static analysis is performed on the source or object code without executing the program. The advantage of this approach is that it offers full code coverage however most memory and concurrency issues can not be detected.

Dynamic analysis is performed while the program is executed. The efficiency and accuracy of the analysis output depends on the test inputs, which determine the code coverage during execution. The program can be started either on a real or a virtual processor. The code is usually examined by instrumenting the client code with analysis code to obtain the necessary information about the run-time state. The main advantage of dynamic analysis is that most issues like memory leaks and concurrency problems can be detected. However, detection of such problems is directly related to program input [5]. Profiling is a subcategory of dynamic analysis.

Profiling needs to gather information, and the most used methods are based on sampling, instrumentation and event based gathering.

The aim of performance optimization of a program is to decrease the resources used. The relationship between performance analysis(profiling) and performance optimization is established by the probability of finding flaws and bottlenecks in the software using a profiler. The profilers aim to inspect specific aspects of the program, such as duration of execution, memory used,

multithreading or I/O operations [12], [14]. Profiling is therefore an important step in optimizing code.

The most important aspects regarding performance analysis are:

80/20 principle: According to the Pareto principle [13], for almost every event, 20% of the causes are responsible for 80% of the effects. The 80/20 rule can also be applied in software optimization. This means that approximately 20% of the code is using 80% of resources, time, memory etc. Identifying this small percentage of code by manually analyzing the source code can be a demanding task and a simpler solution for this problem consists in using a profiler.

Time-based analysis: is very useful in identifying parts of the program which are consuming the most processing time during execution. Measuring the duration alone is not enough because todays applications may also be influenced negatively by cache hit rates, disk read and writes, inter-thread communication and even overhead generated by the operating system.

Analysing the memory allocations and deallocations during the execution of a program can help the programmer finding code which is memory-consuming and therefore affects the overall performance of the application. Another important aspect is finding memory related bugs, like accessing unallocated memory, double freeing or memory leaks which can lead to corrupted data or application crashes.

Multithreading Analysis: Most applications use multiple threads or processes to make use of the processors with more than one core as well as to divide a program intro multiple tasks. The issues that arise from using threads are mostly caused by the shared access to resources. The mechanism for protecting access can lead to deadlocks, livelocks, starvation and unprotected access may cause race conditions

Cache and Branch-prediction Analysis: CPUs use caches that are organized hierarchically in order to improve the performance of accessing the main memory. Each level increases in size but has reduced speed compared to previous levels. Cache hits or misses can have a great impact on software performance. Another important facet of performance analysis is branch prediction. A CPU attempts to predict which branch of a conditional jump instruction will be taken in order be able to continue execution without waiting for the data required for the condition.

Valgrind [15] is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

The core is a low-level implementation that is required to build a dynamic instrumentation tool. It works by simulating a CPU in order to fully supervise the execution of the program. Valgrind implements a JIT compiler for instrumenting the program in real-time.

3. Our approach

In this section we will describe *remgrind*, a tool built for the resource management of C/C++ applications. By resources we are referring to heap memory allocated and deallocated and the bytes that are read or written to handles like files or sockets. Our approach to resource management of a program is a profiling tool that collects information about the resources per function and also provides statistics regarding I/O operations per application.

The supported functions refer to allocation and deallocation (malloc, new, vec new, memalign, calloc, free, delete, vec delete) and the supported read and write system calls with their corresponding numbers are: read 0, pread 17, readv 19,recvfrom 45, recvmsg 47, msgrcv 70, mq timedreceive 243, preadv 295, recvmmsg 299, respectively write 1, pwrite 18, writev 20, sendfile 40, sendmsg 46, sendto 44, msgsnd 69, mq timedsend 242, pwritev 296, sendmmsg 307.

Our solution is build as a Valgrind tool and uses instrumentation, replacement of allocation and deallocation functions and system call wrappers. It uses hash tables from Valgrind and implements algorithms for these three main entry points for gathering information about the functions.

Instrumentation is based on the following approach: it takes as argument a superblock that contains a list of statements which are then analyzed one by one. One assembly instruction is split into multiple IR(intermediate representation) instructions, therefore we need to use the tag IMark to identify the beginning of the machine instruction. We use instrumentation in order to add new nodes to the functions hash table and increment the number of calls for a function. This is performed by the helper C function $\&create\_or\_incrementFn$. The algorithm is sketched in 1.

Replacing allocation and deallocation functions: Allocation and deallocation functions can be replaced in Valgrind and this functionality is very useful for gathering information regarding memory. We use this technique in order to compute the memory used by the function itself and its maximum, as well as the cumulative memory and its maximum.

Computing cumulative memory is based on the following idea: after the program has finished execution this algorithm computes the memory used by a function recursively. If the hash node corresponding to the function has no called functions, then it means that the memory must not be calculated any

Algorithm 1 Instrumentation

```
1: function (SB sb)
2:     SB out = sb.expressions;
3:     for each stmt in sb.statement-list do
4:         if stmt.tag == IMark and stmt == entry then
5:             addStmt(out, &create_or_incrementFn)
6:         else
7:             addStmt(out, stmt)
8:         end if
9:     end for
10:    ret out
11: end function
```

further for that function. Otherwise the sum is computed recursively for every called function.

Computing maximum cumulative memory follows this idea: When a malloc, free or a similar function is called, the tool recalculates the current memory for almost every function. For calculating the maximum value of the memory at any point in any function, we traverse the stacktrace and sum the size of the memory allocated. We ignore the first element in the stack because it is the call to the allocation or deallocation function. The values on the stack are pointers to current instructions in every function, therefore the address of the function it belongs to needs to be computed.

Syscall wrappers: System calls can not be fully replaced by Valgrind in a way similar to malloc or other similar operation, but Valgrind offers syscall wrappers which are called before and after the execution of a system call. We use these wrappers to compute the total number of bytes read and written to and from handles and to obtain information about their duration. The number of bytes read or written are added to the corresponding function before the system call by the function *pre_syscall*.

For measuring the duration of a system call, the time is measured by *pre_syscall* and by *post_syscall* and those values are then substracted. The average duration of a system call as well the maximum system call duration are also computed by *post_syscall*.

3.1. **Example.** In this section we will present an example that covers all the possible use cases. We will not consider the functions that are called before main and their data generated regarding reads, writes and memory. Assume we have the following program:

The statistics per application generated by the application are presented in Figure 1, and the statistics per function are listed in Figure 2:

```
1: function 1
2:     pointer p = allocate_memory(120)
3:     read(filehandler, 4000)
4: end function
5: function (Pointer p)
6:     Pointer p1 = allocate_memory(400)
7:     call function1()
8:     free_memory(p1)
9:     free_memory(p)
10:    write(filehandler, 1300)
11: end function
12: function Main
13:    Pointer p = allocate_memory(40)
14:    call function(p)
15:    call function1()
16:    read(filehandler, 3000)
17:    write(filehandler,70000)
18: end function
```

```
==9394==  TOTAL reads I/O: 7000 bytes
==9394==
==9394==  TOTAL writes I/O: 71300 bytes
==9394==
==9394==
==9394==  AVG time reads I/O: 0 sec 0 ms 4214 ns
==9394==
==9394==  MAX time reads I/O: 0 sec 0 ms 8500 ns
==9394==
==9394==
==9394==  AVG time writes I/O: 0 sec 0 ms 24654 ns
==9394==
==9394==  MAX time writes I/O: 0 sec 1 ms 3563 ns
==9394==
==9394==
==9394== MAX cumulative memory overall: 560 B main
```

FIGURE 1. Statistics per application

4. Related work

4.1. **Similar Tools.** There exists a significant number of tools that address program analysis providing information about different aspects during program execution: error detection, overall execution time, memory information. We discuss some of them, as they target the same issues as *remgrind*.

Memchek is used for detecting errors in C and C++ applications. It is implemented as a Valgrind tool. It can detect illegal memory accesses, use of undefined values, incorrect use of allocation and deallocation functions, like double-freeing of a heap memory block and detect memory leaks. The heap summary shows the total number of bytes allocated, the number of allocations

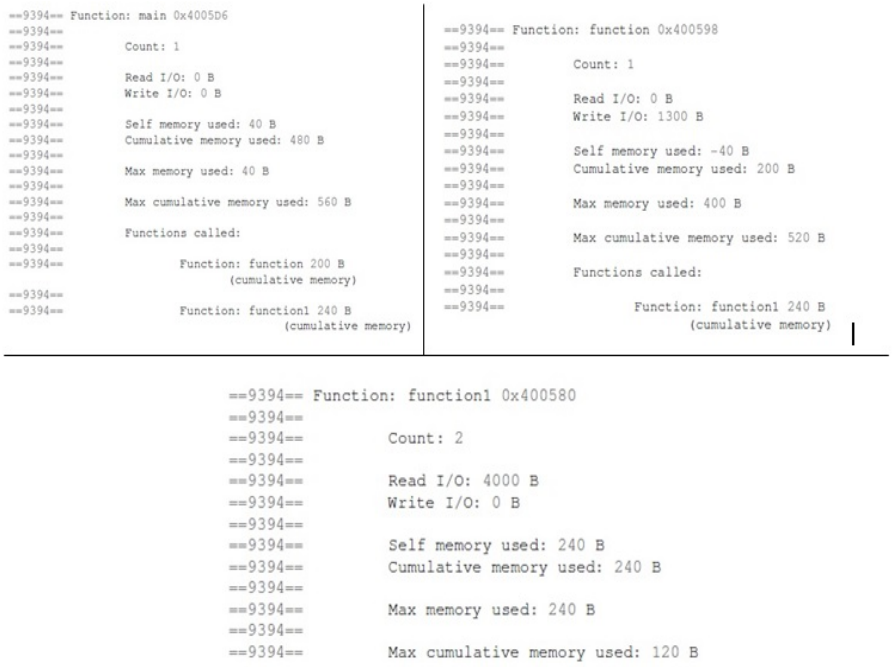

FIGURE 2. Statistics per function

and frees. The leak summary shows the number of bytes lost as well as the stacktrace for every leak [11]

Callgrind is as profiler that generates a flat output that contains information about events like data reads, cache hit or misses for every function of the profiled program [3]. The cost is computed based on this events and can be inclusive or exclusive. Callgrind also generates a cal-graph for the program and in combination with the data from the flat output it can be used to detect the location where the costs are the highest.

Massif is a Valgrind tool for profiling memory. This tool uses snapshots to show the memory usage over time in order to help fixing certain memory leaks that memcheck is unable to detect. It also notes the peak of memory usage and provides stacktraces at certain points in time to show where memory is being used. Massif is able to track heap memory and optionally stack usage or is able to track memory usage per page using lower level interfaces [10].

GProf is the GNU profiler that measures the time spent in functions, the number of function calls and calls per line. It generates a flat profile that contains information about the execution time of the function and the number of calls and it shows the percentage of that function uses. The call-graph it

generates can be used to detect if the time spent in the function was not actually executed in that function but rather in its children, which are the functions it has called [8, 7].

4.2. **Performance Analysis.** In this section we discuss the performance of different tools when analyzing two test programs. The first is a simple program that calls twice a function that opens and reads a file. The second program is an application of allocation, freeing and allocating some memory zone. The programs were compiled using gcc, version 4.8.2 20131212 (Red Hat 4.8.2-7). The platform on which the tests were run is a Laptop with a Intel(R) Core(TM) i5 CPU M 460 @ 2.53GHz processor and 4GB of RAM. The OS is Fedora 20 and the filesystem used in for the first test is ext4.

The test programs were compiled with gcc and the -pg option was added in order to profile using gprof. The results of the testing can be seen in Table 1. Real refers to the time from start to finish of the call, user is the amount of CPU time spent in user-mode code (outside the kernel) within the process, and sys is the amount of CPU time spent in the kernel within the process.

Tool	Test1	Test 2	Details
native	0m0.002s	0m0.004s	real
	0m0.001s	0m0.003s	user
	0m0.000s	0m0.001s	sys
gprof	0m0.001s	0m0.004s	real
	0m0.001s	0m0.003s	user
	0m0.001s	0m0.001s	sys
memcheck	0m0.395s	0m0.483s	real
	0m0.358s	0m0.441s	user
	0m0.033s	0m0.036s	sys
callgrind	0m0.623s	0m0.305s	real
	0m0.216s	0m0.276s	user
	0m0.029s	0m0.025s	sys
massif	0m0.782s	0m0.265s	real
	0m0.203s	0m0.233s	user
	0m0.028s	0m0.028s	sys
remgrind	0m0.229s	0m0.474s	real
	0m0.205s	0m0.445s	user
	0m0.022s	0m0.026s	sys

TABLE 1. Tool Performance

The unique features of *remgrind* compared to the tools we have analyzed and how they can be used to improve software can be summarized in the following:

Self memory used per function: This value is useful in detecting functions that allocate or deallocate a lot of memory without considering the functions it called. The difference compared with memcheck is that this is not intended as a memory error detector but resource management per function.

Maximum value of the self memory used per function: This value can help detect possible peeks in the memory used in a function even if at the end of the function the self memory used has a value of 0. Massif is a heap profiler that can detect peeks in the memory usage but it only provides information about the whole program and is designed to detect memory leaks.

Maximum cumulative memory used per function: This value represent the maximum value of the memory used by a function at any point in time taking the functions it called into consideration. These values can help to detect memory-consuming parts of the program and are 100% accurate, because every time a function allocates or deallocates memory, all the functions from the stack are updated. Massif detects the maximum of the memory used per application, whereas our solution detects maximum per function.

Total reads and writes per function and application:None of the presented tools compute the number of bytes read and written from handles per function or per application. Performance hotspots caused by I/O operations can be easily detected by knowing this information about functions.

Average time spent in I/O per application and the maximum duration of a I/O operation: The average time spent in I/O operations can help in detecting if the performance of the application is I/O bound or not. A tool that is also focused on collecting information per function is callgrind, but it does not offer information about memory usage or I/O operations. The information we provide regarding memory can not be obtained from memcheck or massif, because they are application oriented and not function oriented.

5. Conclusions

In this paper we have presented an approach to resource management of C/C++ applications that has not been previously attempted. It is based on analysis of resources used by an application at the function level in order to be able to determine performance hotspots. The approach is implemented in a tool, *remgrind* that monitors memory allocations and deallocations, and I/O operations for C/C++ applications and annotates each function with this information. We compare our approach with similar tools that address the same issues: we have perform a time performance analysis of *remgrind* and other tools, and then discuss the innovative features of our tool.

Our solution can currently profile single-threaded applications. In the future we want to add support for multi-threaded applications. Another improvement that we want to implement in the future will deal with replacement

of the C helper calls added at instrumentation time for gathering information, with intermediate representation statements, which would lead to a significant increase in performance.

References

[1] Anita Borg, R.E. Kessler, Georgia Lazana, and DavidWall. Long Address Traces from RISC Machines: Generation and Analysis, Proceedings of the 17th Annual Symposium on Computer Architecture, May 1990

[2] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. MIT/LCS/TR-516, MIT, 1991.

[3] Callgrind - Valgrind page callgrind manual [Online]. Available: http://valgrind.org/docs/manual/cl-manual.html

[4] E. Coppa, C. Demetrescu, I. Finocchi - Input-Sensitive Profiling, Proc. PLDI12, , ACM SIGPLAN Notices, Volume 47 Issue 6, June 2012, pg. 89-98

[5] E. Dustin, T. Garrett, and B. Gauf, Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality, 1st ed., Addison-Wesley Professional, March 2009

[6] Susan J.Eggers, DavidR. Keppel, Eric J. Koldinger, andHenryM.Levy.Techniques for Efficient Inline Tracing on aShared-MemoryMultiprocessor. SIGMETRICS Conference on Measurement and Modeling of Computer Systems, vol 8, no 1, May 1990.

[7] Gprof - Gnu gprof profiler [Online]. Available: https://www.cs.utah.edu/dept/old/ texinfo/as/gprof.html

[8] S.L. Graham, P.B. Kessler, and M.K. McKusick, gprof: a Call Graph Execution Profiler, Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 17, No 6, pp. 120-126

[9] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Software, Practice and Experience, vol 24, no. 2, pp 197-218, February 1994.

[10] Massif - Valgrind page massif manual [Online]. Available: http://valgrind.org/docs/manual/ms-manual.html

[11] Memcheck - Valgrind page memcheck manual [Online]. Available: http://valgrind.org/docs/manual/mc-manual.html

[12] S. Meyers, More Effective C++: 35 New Ways to Improve Your Programs and Designs, 1st ed. Addison-Wesley, 1996, pg. 82-85

[13] Pressman, Roger S.: Software Engineering: A Practitioner's Approach (7th ed.). Boston, Mass: McGraw-Hill, 2010

[14] M. Reddy, API Design for C++, 1st ed. Morgan Kaufmann, February 2011, pg. 235-236

[15] Valgrind - Valgrind home page [Online]. Available: http://valgrind.org/

[16] Static code analysis. [Online]. Available: http://www.viva64.com/en/t/0046/

Babeş Bolyai University, Department of Mathematics and Computer Science, 1 M. Kogălniceanu St, 400084 Cluj-Napoca, ROmania

E-mail address: hanneloremarginean137@gmail.com

E-mail address: motogna@cs.ubbcluj.ro