A Tracking and Visualizing System of Memory Usage along to C Source Programs

Kyoko Iwasawa
Computer Science dept.
Takushoku University
Tokyo, Japan
kiwasawa@cs.takushoku-u.ac.jp

Takuhiro Okamuira
COMSOFT Co, Ltd.
Tokyo, Japan
shnfkrm@live.jp

***Abstract*— Our tracking system shows the situation of memory usage of C programs. Its information includes both stack area of local data and dynamic allocated area. The system shows the results off line. First of all, it inserts recording statements to the memory allocation and release of user programs in order to make a log file of memory usage. After the execution of modified user program, the system analyzes the log file to make the usage graph. Based on the graphical image, the user can find out where each memory event occurred on the C source program interactively. Therefore, the user can recognize the accurate location where the largest memory area was used, and find which memory allocation caused memory leak. These functions are efficient for embedded system, whose memory size is strictly limited. In this work in progress, we are attempting to show where user should insert *free* function call by using static data flow analysis.**

Keywords-C source program; memory usage; memory leak; tracking; visualizing;

I. INTRODUCTION

C program developers have to be concerned with the situation of memory usage of their programs. The amount of memory usage is influential in performance of the program, because using large memory often causes cache miss and paging. Also, amount size of memory usage is essential for embedded system [2]. However, it is difficult to know how much memory is necessary and where it should be decreased.

Although, generally, it is difficult to know that when memory usage becomes maximum size and whether there is leak area, this information is necessary to optimize C programs. Programmers need to know the relationship between source code and memory usage. There are some memory management tools for linux. One of them is valgrid [3], which is the multipurpose code profiling and memory debugging tool. It shows whether the memory leak occurred and finds invalid pointer use [4]. It shows the leaked memory address with process ID, so the users would have to look for that address on the allocated memory address list. And it has lots of functions, but does not mention with the size of memory usage.

Our system shows graphically the amount of memory usage in chronological order. For each point on the usage graph, user is able to know the source corresponding source statements interactively. In addition, the system corresponds dynamic allocated memory (i.e., *malloc*() and *calloc*() call) to its release (i.e., *free*() call) [1].

Section II describes the tracking and visualizing system and its output images. Section III shows detail of the log file. Section IV presents the algorithm to calculate the dynamic allocated area, and Section V concludes and describes the future work.

II. OUTLINE OF THE SYSTEM

In order to measure accurately the size of memory usage, our system analyzes executing logs offline. Therefore, the system parses the target C programs and inserts output statements to record of memory events.

A. *Process of Tracking and Visualizing*

Figure 1 shows the analyzing process. First of all, the system picks up statements which cause memory allocation and release. The picked up statements are following four kinds of statements.

(1) Entry point of functions
(2) Return statement in functions
(3) Invoke *alloc* function
(4) Invoke *free* function

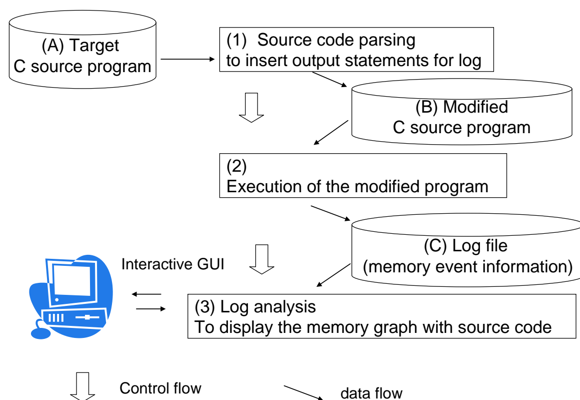


Figure 1: Process of the System

At point (1) and the point (2) the system inserts our prepared function call in order to write the size of local data on stack area, and to write when these local data are

released. At point (3) and the point (4) the system replaces the library call with our prepared function call, which writes the information of dynamic allocated data to the file. Then the modified C source program (Figure 1: (B)) is generated. After the execution of the modified program the system can get the log file which includes the memory usage information. Finally, memory usage graph is displayed and users are able to know details of memory information.

B. *Output of the System*

After the system analyzes the log file, it visualizes the memory usage of C program. By restructuring the log data, memory usage graph is displayed off line as Figure 2. The upper part of the figure shows behavior of stack area usage, and the lower part of the figure shows the behavior of dynamic allocated heap area. Each bar means that the four kinds of memory events, which were described in the previous section and denoted by (1), (2), (3) and (4), occurred. The length of bar shows the size of memory used by program at that point.

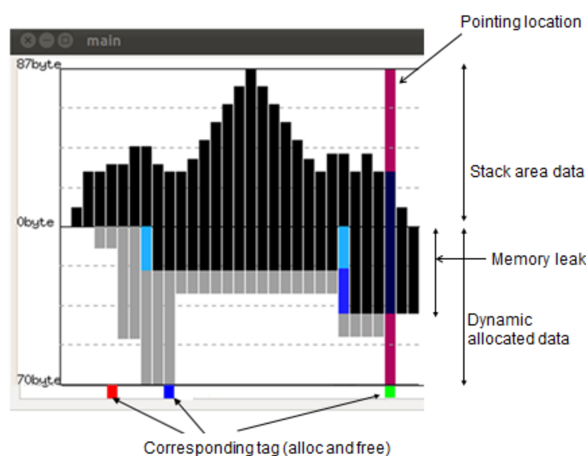
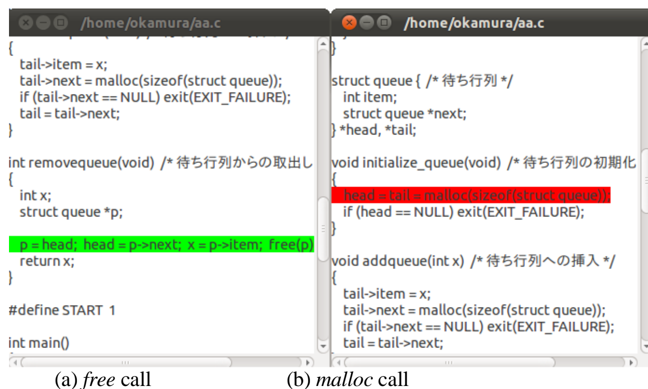


Figure 2: Memory usage graph



(a) *free* call (b) *malloc* call

Figure 3: Source program

When user function is invoked, the size of local data area is added to usage size, and when return statement executes, these local data area is released, so the length of bar is shortened. When the C program has finished, its local data area is always cleared. On the other hand, heap area data, which was allocated by *alloc* function call, is released only by *free* function call. Consequently, if there is not enough *free* function call, memory leak will occur when program has finished.

When user clicks on a bar of the graph in Figure 2, the system shows the source program which caused this event. Figure 3 (a) shows the *free* function call of the source program, which corresponds to the clicked bar in the lower part. It shortened because the *free* function call releases memory area. A bar in the lower part lengthens, when *malloc* function call allocates memory area dynamically, as shown in Figure 3 (b). At the end of the execution, a bar in the lower part means memory leak. User might insert *free* function call, because the user can find out where this area is allocated and its identifier from the tags on the bottom of the graph (as seen in Figure 3:). If *free* function call released linked area (linked list or tree or graph structure, etc.), the system founds all *alloc* function call statements. The small tags on the bottom of the graph mean the correspondence *free* call to *alloc* call.

The bar in the upper part increases when user function call invoked, and then it shortens by return. As these bars in the upper part mean the stack area, when a program has finished, it always becomes zero. The total length of a bar means the size of memory, which user program was using at that point.

III. DATA IN THE LOG FILE

The detail of the logging data (shown as Figure 4:) is described in this section. In order to measure accurately the size of memory usage, user program is parsed and modified. User program is inserted the system function call to log the memory information.

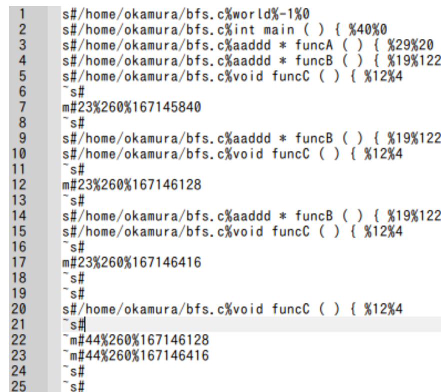


Figure 4: Log file

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9

A. Local Data on the Stack Area

The size of local data on the stack area is accumulated the sum of all sizes of local data (variable, array, etc.), which are caused by nested user function call.

(1) Entry point of functions

The system inserts the function call to output file following five items to the log file.

- Tag ("s" means entry point of function)
- Directory path and file name
- Function name
- Line number of entry point of function in Source program
- Sum of local data size (byte) from static data declarations

(2) Return points of functions

When return statement is executed, system records the tag, which shows that the program returned from the function and its stack area data is released.

- Tag ("~s" means return point of function)
- Directory path and file name

B. Dynamic Allocated Data

The size of dynamic allocated data on the heap area is recorded by each *alloc* (*malloc* and *calloc*) function call and *free* function call. As a result, we can get the log file as Figure 4 shows. It includes the accurate and detailed memory usage situation. Each record is kept in the chronological order.

(3) Invoke *malloc* and *calloc* function

- Tag ("m" means *alloc* function call)
- Line number of *alloc* function call
- Size of allocated area (actual parameter of *alloc* call)
- Address of allocated area (return value of *alloc* function call)

(4) Invoke *free* function

- Tag ("~m" means *free* function call
- Line number of *free* function call
- Size of released area (system find out its own data → sec.4)
- Address of released area (actual parameter of *free* function call)

IV. CALCULATION OF DATA SIZE

Although we would like to know the leaked memory size and where the leaked memory was allocated on source code, some information of dynamic allocated data was lost when the program was finished. Therefore, the system function has to collect information while user program is running.

In the case of linked structure (list, tree and graph, etc.), it is difficult to accumulate released area by *free* function call. The parameter of *free* function is the pointer of the data, so there is no information as to the size of the data. Furthermore, *free* function released all of linked data, so that the system has to keep allocated memory information until it is released by using linked list that we call "Alloc List".

A. Alloc Function Call

Our system function $ALLOC, which is invoked instead of the original *alloc* function, has the following three tasks.

☐ To allocate memory by calling original *alloc* function

☐ To make the cell of Alloc List including the address and the size

☐ To write the log file (tag, address, size, line number of source program)

This function is given line number of source program and the data size to allocate as formal parameters. Figure 5 shows the process of $ALLOC. The system replaces the *alloc* function call to $ALLOC function call, and links $ALLOC function to modified program.

The cell of Alloc List has the address which is returned by *alloc* function and allocated memory size (byte), which is actual parameter of *alloc* function. Each cell is connected in chorological order as Alloc List.

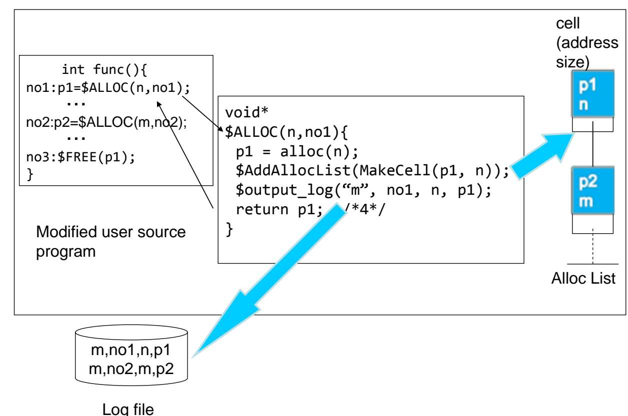


Figure 5: $ALLOC function and Alloc List

B. Free Function Call

Our system function $FREE, which is invoked instead of the original *free* function call, has following the four tasks.

☐ To find address given as formal parameter from Alloc List.

☐ To write log file (tag, address, size, line number)

☐ To search linked area to release recursively from Alloc List

☐ To release area by calling original free function

This function is given the address that is at the top of release area, and line number of source code as formal parameters. Figure 6 shows the process of $FREE function. The system replaces the *free* function call by the $FREE function call, and links $FREE function to modified program.

The system finds the address, which is the actual parameter of original *free* function call, from Alloc List, and

Copyright (c) IARIA, 2013.

writes the line number, size, and address. Then, the system has to find any linked area which would be released by the same *free* function call, so it searches any allocated address between given address and given address + size. If system finds out the address that is kept in Alloc List, it will search the next linked area recursively. While finding each area, system gets which alloc function call corresponds to this *free* function call.

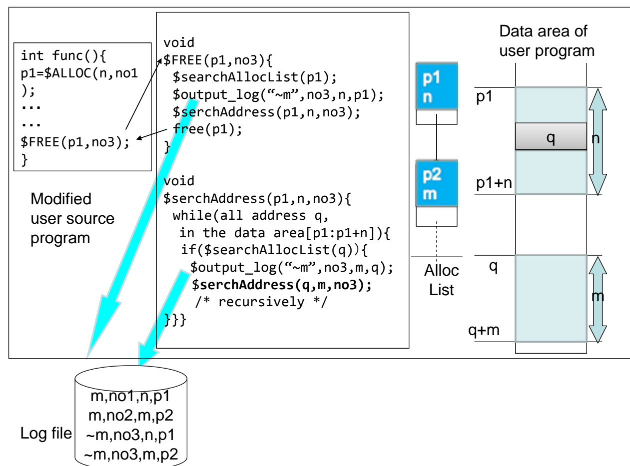


Figure 6: $FREE function and Alloc List

V. CONCLUSION AND FUTURE WORKS

Our tracking and visualizing system shows the situation of memory usage of C programs. It includes both stack area of local data and dynamic allocated area. It makes a log file about memory usage by modified user program execution, and then analyzes the log file to make the memory usage graph. Based on graphical image, the user can find out where each event occurred on the C source program interactively.

Therefore, the user can recognize the accurate location where the largest memory area was used on source program and which memory allocation caused memory leak. It is efficient to optimize the usage memory size of C programs, because usually finding out the memory size on each line is very difficult. These functions are efficient for embedded system, whose memory size is limited strictly. The system just begins working and we could try some small test programs. These programs make and modify queue structure and binary tree structure. We have to evaluate the system performance by using more practical programs. More information is necessary for users to optimize the program.

For the future work, we would like to show the last access of dynamic allocated memory. This is the reason why it means the earliest point that the allocated memory can be release. We are considering using static data flow analysis and static type analysis. Tracking only uncertain reference as a result of static analysis, we can reduce log data and analyzing time.

REFERENCES

[1] B. W. Kernighan and D. Ritchie, The C Programming Language (2nd Edition) , Prentice Hall, 1988.

[2] CoActionOS, [http://www.coactionos.com/embedded-design/101-understanding-memory-usage-in-c.html](Aug. 2013)

[3] [http://www.valgrind.org](Aug. 2013)

[4] A. Allain, "Using Valgrind to Find Memory Leaks and Onvalid Memory Use", [http://www.cprogramming.com/debugging/valgrind.html](Aug. 2013)

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9