Aspect-Oriented Tool For Memory And Performance Profiling

Ankit Agarwal[1], Shivanjali Kamble[1]
Indian Institute of Technology Rajasthan, India
Email: ankitgenius90@gmail.com, ksshivanjali@gmail.com

***Abstract*—With the motivation of improving the performance of softwares, we are presenting an Aspect-Oriented profiling tool, termed as *AspectTrace*, which can profile any C code. Accurate, efficient and dynamic profiling tools are needed to study the program execution behaviour. This information is then used by the programmers and developers to optimize the programs. We have proposed and implemented a new tool *AspectTrace*, a profiler and memory-leaks detector, that performs dynamic program analysis. It consists of a loadable kernel module, which extracts the kernel level information as per instructed by automatic generated aspects. The proposed aspect-oriented approach used for profiling is different than the present approaches (statistical, virtual machine) used by existing profilers. The aim of this approach is to produce streamlined profiler code, fine-grained profiling, less runtime overhead, architecture independence, user-friendliness and ease to use.**

I. Introduction

From the very first day when operating systems were developed, performance improvement has always been a concern. Amdahl's law indicates that in order to write fast and efficient program, programmers must target the most expensive part of their code[1]. The significant part is the detailed study of the behavior of a program which can help to figure out the pinpointed sections of the code which should be optimized. Profiling is a standard method to investigate and tune the performance of any code. A naive and most popular way of profiling is to manually edit several statements so that the time-stamp will be printed when the thread of control reaches one of those statements. But the major drawback of the manual editing is that it is error-prone. Moreover, the analysis of the large code with manual editing is a cumbersome task. Profilers are the tools which help in profiling the code. They are developed mainly to figure out which part of the code consumes the major portion of the execution time and the part of the code where memory leaks occur. Different kind of profiling tools have been developed till date. Some work at the user level and some at the kernel level. For kernel level profiling, there are various profilers available but they are not able to do fine-grained profiling, have high CPU overhead, provide limited information, can only profile specific types of activity, or rely on kernel instrumentation. Many other new techniques have been introduced in this field such as the concept of latency[2] and dynamic aspect-oriented methodologies[3]. We are introducing a new profiler and memory-leaks detector, *AspectTrace*, that performs dynamic program analysis.

[1]Student Authors

II. Our Approach

We propose an architecture for the profiler, *AspectTrace*, based on solely new approach, Aspect-Oriented approach. Aspect-Oriented programming includes the programming techniques which increases the modularity by allowing the separation of concerns. It breaks down the program logic into distinct parts. The crosscutting expressions perform the grouping and encapsulation of concerns in one place. Pointcuts describe a set of join points by determining the condition on which an aspect shall take effect. Thereby each join point can either refer to a function, an attribute, a type, a variable, or a point so that this condition can be for instance the event of reaching a designated code position. Depending on the kind of pointcuts, they are evaluated either at compile time or at runtime[4].

Aspect-Oriented programming is an API. Aspects have been written to have logging code defined along with their position of execution. The integration of any program code and aspects is done through a process known as weaving. An aspect-weaver reads the aspects and generates the corresponding object-oriented code with the aspects weaved into it. The weaved aspects become active whenever the join points get executed. Any change in the functionality of a function or module for which an aspect has been written would not affect the functionality of the aspects and vice-versa.

AspectTrace works on the same principle and uses aspect-oriented programming as a primary tool. When the target program is to be profiled, the aspects are generated for it and are weaved to it to generate a single C++ file, which is then compiled using the aspect compiler. The weaving of aspects and the compilation of the generated code are the subsequent processes of the same tool. The aspects are generated for each function in the target program, performing the pre-defined tasks such as displaying the virtual memory allocation, memory leaks, the execution time of the program and the function analysis of the program.

A. Architecture

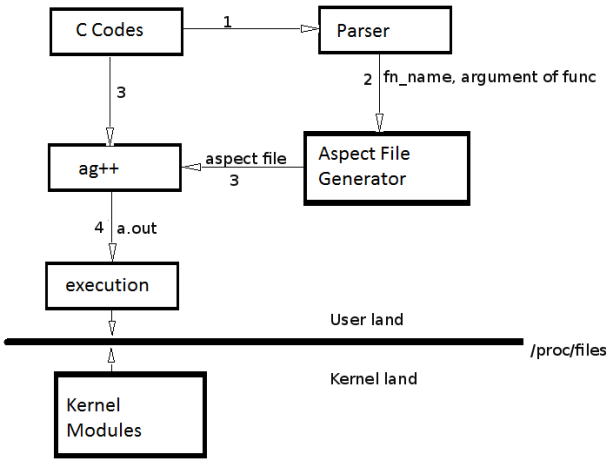

Fig. 1: AspectTrace Architecture

The architecture is divided into two parts, viz., user land and kernel land. The user land consists of *parser*, *aspects generator* and *aspects weaver*. The target code is parsed to extract the names of the functions defined in the target program, and these functions' names along with their parameters are stored in a file. This file is then used to generate aspects from the pre-defined aspect generator. The generated aspects are then weaved to the target code using the aspect weaver to give a weaved file. The aspect weaver weaves the aspect code to the main code. The weaver used for the purpose is ag++. ag++ is an aspect compiler which compiles and integrates both the aspect code and target code. The weaved, compiled file is then executed to perform the profiling.

In the kernel land, *prof.ko* module is loaded in the linux kernel 2.6.32. The */proc* filesystem has been used as an interface between the user land and the kernel land. Two */proc* files are created when the module is loaded into the kernel: */proc/main* and */proc/func*. The former file calls the module which is specific for the *main()* function's aspects and the later one deals with the module for the user-defined functions' aspects. Whenever, the */proc* file is written/modified, the corresponding module becomes active and performs the assigned tasks.

B. Software Details

The user land performs the functional analysis and the memory-leak detection of the target code whereas the kernel land extracts the information about the virtual memory allocated to the program.

The phenomenon of memory leaks and execution time has been implemented in the aspects itself. The memory leaks' joinpoints get invoked whenever the dynamic memory allocation functions, viz., *malloc* or *calloc* or *realloc* or *free* are called. They calculate the amount of memory allocated and memory freed for each function. The details of the dynamic memory allocation: the line number in the program at which the memory allocation is called, the memory location at which the memory is allocated in the virtual memory and the size of the allocated memory are stored in a file. This file is used to display the memory leaks at the end of execution. The execution time taken by the code and the aspects are recorded separately by the aspect. Moreover,

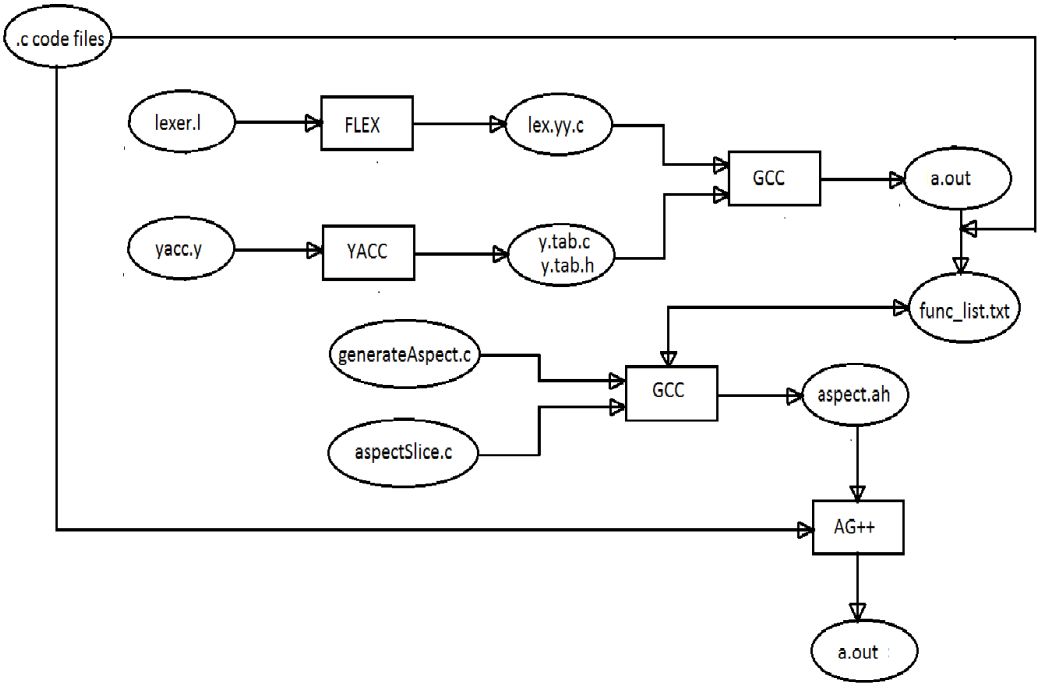

Fig. 2: Code Flow Graph

the execution time for each function is calculated separately. This provides the execution overhead of each function (excluding aspects overhead).

The loaded module traverses the process list in the kernel using the *task_struct* struct defined in *linux/sched.h* header file. The module uses the process id, *pid*, obtained from */proc* files, to find the corresponding *task_struct* pointer. The *mm_struct* struct pointer is then used to fetch the memory details for the process. This struct gives a pointer to struct *vm_area_struct* defined in *mm_types.h*. The *vm_area_struct* contains all the information about the virtual memory allocated to the process. This struct gives all the required details about the virtual memory block assigned to the program, such as, the start and end address of the block, file associated with the block and the permission flags. Thereby, the loaded module extracts the information about the amount of memory allocated to various memory segments, viz., text segment, data segment, heap and stack.

III. Result and Analysis

AspectTrace is a generic, kernel-level, dynamic analyser which can be used to study any C program behavior in detail. It provides various information about the code profiled :

a) Functional Analysis : functions that are executed, the call graph, execution time of each function, each function's overhead for the program.
b) Memory Details : total virtual memory allocated for the program, starting address of text section, size of text section, starting address of data section, size of data section, starting address of heap and size of the total heap allocated, starting address of stack, memory leaks and their details (location, amount).
c) Other Details : process id, aspect overhead. *AspectTrace* has much less runtime overhead. It is accurate and efficient.

There exist many other profilers such as *gcov*, *gprof* and *valgrind* which are widely used for profiling. All of them provide the dynamic analysis of the program. A comparative study of AspectTrace has been done with these profilers.

A. Output-wise Analysis

gcov provides the information about the frequency of the execution of each line in the code[5]. *gprof* provides information about the frequency of the execution of the functions along with their execution time as well as gives the callgraph of functions[6]. *valgrind* uses its own

virtual machine to do the profiling and its *memcheck* tool tells about the memory leaks and the amount of heap allocated in the program[7]. The outputs of these profilers for a matrix multiplication program are provided in annexure. All the codes have been executed on linux kernel 2.6.32 and Intel CORE 2 Duo Processor. The output produced by *AspectTrace* is consistent with the result produced by these profilers.

B. Approach-wise Analysis

gprof works on the sampling approach, i.e., it performs the periodical probing of the target program's program counter. The resulting time values are thus a statistical approximation and not accurate, whereas *AspectTrace* calculates the time values at the entry and exit points of the functions, thus it gives the accurate execution time of the functions. The *memcheck* tool of the *valgrind* profiler replaces the original memory allocation code with its own virtual machine implementation. It inserts an extra instrumentation code around all the instructions. These result in the high execution time of the program. On the other hand, the join points corresponding to the memory check in *AspectTrace* gets invoked only when the memory allocation functions are invoked, resulting in the significant reduced overhead of the profiler.

C. Time Analysis

TABLE I: Execution Time of Profilers (in sec.)

Gcov	Gprof	Valgrind	AspectTrace
61.5	60.9	479.4	61.8

The total time taken by the profilers to compile, profile and execute the matrix multiplication program is compared in Table I. In this program, two 1500x1500 matrices are randomly initialized and then multiplied to give the product matrix. The time taken by *gcov*, *gprof* and *AspectTrace* are almost same. The information provided by *gcov* and *gprof* is limited whereas *AspectTrace* provides vast information presenting various kinds of details. *Valgrind* on the other hand, has the higher profiling time due to the runtime instrumentation. The time taken by *AspectTrace* is about 8 times less than the time consumed by *valgrind*.

IV. Conclusion and Future Work

The Aspect-Oriented approach is solely new. It has strength to capture small details with ease thus leading to fine-grained, smooth profiling, with significantly low overhead. Exploring this capability can lead to significant improvement in the world of profiling. Our tool, *AspectTrace* has much less runtime overhead. It is accurate, efficient and provides vast information about the program execution. This tool has high potential in High Performance Computing. In future we can even think, to extend *AspectTrace* profiler to profile the Linux kernel. One can add more elements of profiling to extract small things as per the requisites and do fine-grained profiling. Fullfledge *AspectTrace* can be developed including many tools such as memory tools detecting : use of uninitialized memory, reading/writing memory after it has been freed, and reading/writing off the end of malloc'd blocks.

V. Acknowledgement

The authors would like to thank Dr. Anupam Gupta, Project Officer, Indian Institute of Technology Rajasthan, India for initiating the idea on this new approach in the field of profiling.

References

[1] Raj Jain, "Survey of Software Monitoring and Profiling Tools", Washington University, 2006 [Online] http://www.cse.wustl.edu/∼jain/cse567-06/ftp/sw_monitors2/index.html

[2] Nikolai Joukov, Avishay traegar, Rakesh iyer, Charles P. Wright and Erez Zadok, Operating System Profiling via Latency Analysis, 7th Symposium on Operating Systems Design and Implementation (OSDI 2006).

[3] Yashishato Yanagisawa, "A Source Level Kernel Profiler based on Dynamic Aspect-Orientation, Tokyo Institute of Technology, 2005

[4] "AspectC++ Language Reference" [Online] Available at: http://www.aspectc.org

[5] "gcov manual" [Online] Available at: http://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Gcov-Intro.html

[6] Jay Fenlason, "GNU gprof manual", 1988

[7] "Valgrind User Manual" [Online] Available at: http://valgrind.org/docs/manual/mc-manual.html

VI. ANNEXURE

A. *Outputs of different Profilers*

Gcov

```
    1:   83:void calculate_matrix(int *matrixA,int rowA,
                 int colA,int *matrixB,int colB,int *matrixC)
    -:   84:{
    -:   85:    int i,j,k;
    -:   86:
 1501:   87:    for(i=0;i<rowA; i++)
    -:   88:    {
 2251500:   89:        for(j=0;j<colB;j++)
    -:   90:        {
377250000:   91:            for(k=0;k<colA;k++)
    -:   92:            {
375000000:   93:                *(matrixC+(i*colB)+k) +=
*(matrixA+(i*colA)+k) * (*(matrixB+(k*colB)+j));
    -:   94:            }
    -:   95:        }
    -:   96:    }
    -:   97:
    -:   98:    //display_matrix(matrixC, rowA, colB);
    1:   99:}
```

Gprof

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
99.93	59.05	59.05	1	59.05	59.05	calculate_matrix
0.07	59.09	0.04	2	0.02	0.02	insert_matrix

Valgrind

```
==9103== HEAP SUMMARY:
==9103==     in use at exit: 9,000,000 bytes in 1 blocks
==9103==   total heap usage: 3 allocs, 2 frees, 27,000,000 bytes allocated
==9103== 9,000,000 bytes in 1 blocks are possibly lost in loss record 1 of 1
==9103==    at 0x4024F20: malloc (vg_replace_malloc.c:236)
==9103==    by 0x8048679: main
==9103==
==9103== LEAK SUMMARY:
==9103==    definitely lost: 0 bytes in 0 blocks
==9103==    indirectly lost: 0 bytes in 0 blocks
==9103==      possibly lost: 9,000,000 bytes in 1 blocks
==9103==    still reachable: 0 bytes in 0 blocks
```

AspectTrace

```
Process id: 16575
Memory Design: NUMA


VIRTUAL MEMORY INFO:
Start Code:0x8048000
End Code:0x804b95c
Code size:14684

Start Data:0x804cef4
End Data:0x804d064
Data size:368

Start Heap:0x8ed5000
End Heap:0x8ef6000
Heap size:135168

Start Stack:0xbfbd8da0
Total virtual memory allocated:3002368


CALLGRAPH :
Enters function: main()

Entered function: insert_matrix()
Total virtual memory allocated:30015488
Exits function: insert_matrix()
Total virtual memory allocated:30015488
Execution time of insert_matrix() : 0.081902
Execution time of insert_matrix() with aspects : 0.082260

Entered function: insert_matrix()
Total virtual memory allocated:30015488
Exits function: insert_matrix()
Total virtual memory allocated:30015488
Execution time of insert_matrix() : 0.090835
Execution time of insert_matrix() with aspects : 0.091060

Entered function: calculate_matrix()
Total virtual memory allocated:30015488
Exits function: calculate_matrix()
Total virtual memory allocated:30015488
Execution time of calculate_matrix() : 60.596458
Execution time of calculate_matrix() with aspects : 60.596695

Exited Function : main()
Execution time of main() : 60.772617
Execution time with aspects : 60.773273

Aspects Overhead : 0.000656
Percentage Overhead : 0.001079%


FUNCTION ANALYSIS :
insert_matrix       0.081902    0.134776%
insert_matrix       0.090835    0.149475%
calculate_matrix    60.596458   99.715749%


HEAP ALLOCATED FOR PROGRAM : 27000000 bytes


MEMORY LEAKS :
9000000 bytes
Location of memory leaks :
0xb5edd008 9000000 29
```