COMPILER DESIGN

(UE15CS351)

CALL SEQUENCE GENERATION

Authors

Aditya Jain (01FB15ECS018)

Anuj Bhushan (01FB15ECS044)

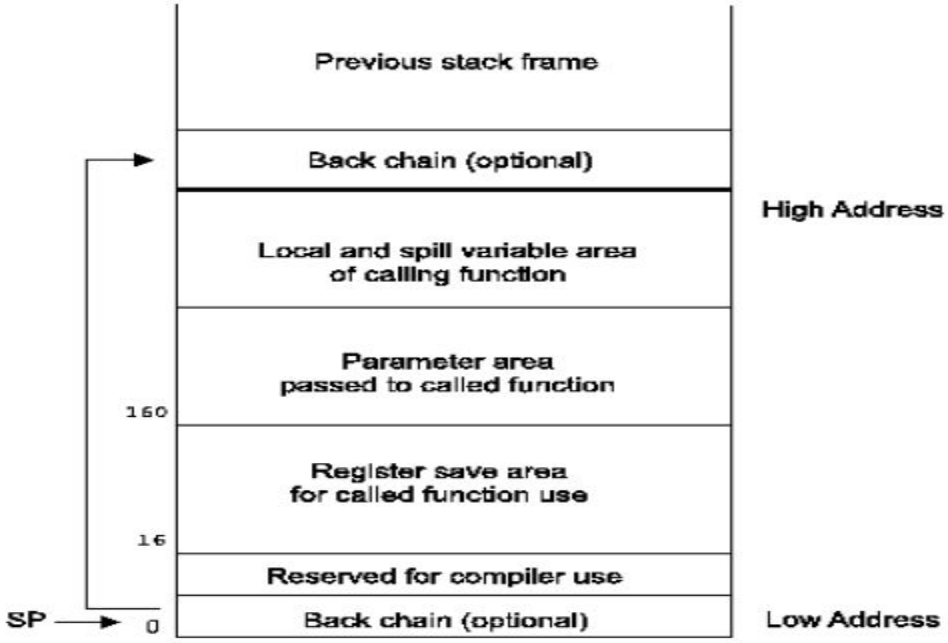
INTRODUCTION

In compiler , a calling sequence is an implementation-level (low-level) scheme for sequence of instructions that is used to call a subroutine. Differences in various implementations include where parameters, return values, return addresses and scope links are placed, memory used ,and how the tasks of preparing for a function call and restoring the environment afterward are divided between the caller and the callee. Although some languages actually may specify this partially in the programming language specification, different implementations of such languages (i.e. different compilers) may typically still use various calling conventions, often selectable. Reasons for this are performance, frequent adaptation to the conventions of other popular languages (with or without technical reasons), and restrictions or conventions imposed by various "platforms" (combinations of CPU architectures and operating systems).

Calling Sequence Implementation

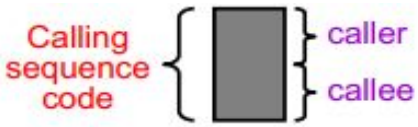
Call graphs can be dynamic or static. A dynamic call graph is a record of an execution of the program, for example as output by a profiler. Thus, a dynamic call graph can be exact, but only describes one run of the program. A static call graph is a call graph intended to represent every possible run of the program. The exact static call graph is an undecidable problem, so static call graph algorithms are generally over approximations.That is, every call relationship that occurs is represented in the graph, and possibly also some call relationships that would never occur in actual runs of the program.

A function will be passed a frame on the runtime stack by the function which called it, and may allocate a new stack frame. A new stack frame is required if the called function will in turn call further functions (which must be passed the address of the new frame). This stack grows downwards from high addresses. Figure below shows the stack frame organization. SP in the figure denotes the stack pointer (general purpose register r15) passed to the called function on entry. Maintenance of the back chain pointers is not a requirement of the ABI, but the storage area for these pointers must be allocated whether used or not.



Calling Sequences: Division of Responsibilities

The code in a calling sequence is often divided up between the caller and the callee



If there are m calls to a procedure, the instructions in the caller's part of the calling sequence is repeated m times, while the callee's part is repeated exactly once

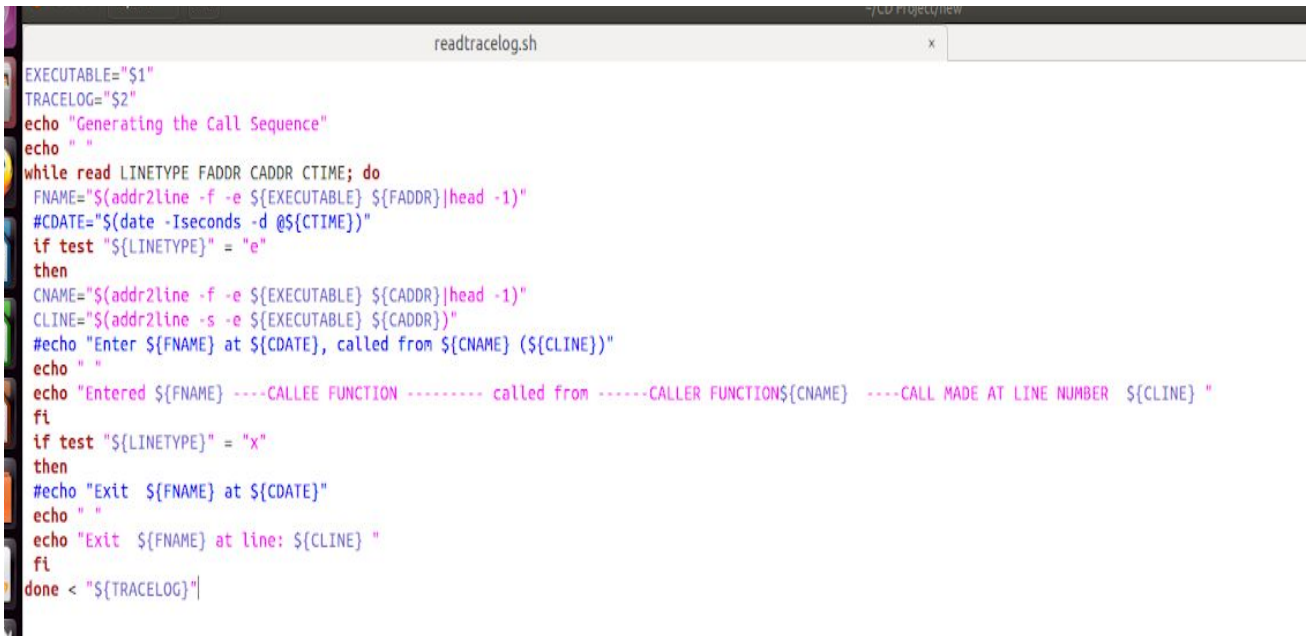
- – This suggests that we should try to put as much of the calling sequence as possible in the callee.
- – However, it may be possible to carry out more call specific optimization by putting more of the code into the caller instead of the callee.

Implementation Details:

Our Implementation for generating the call sequence starts with analyzing the input file. We analyze the control flow of the source program and understand where the call is happening and important attributes like the caller and the callee.

We have a user program for which we need to generate the call sequence. Our implementation makes the user program to be run in such a way that no changes will be made in the user program and the program's function call sequence gets generated with the help of another program that was executed with it.

The other program monitors the debugger file and traces all the function calls and puts them in a log file. The Log file only maintains a history of the address shifts being made in the execution sequence and from the log file we are able to extract the Addresses where the function were being called and then we print them in the terminal.

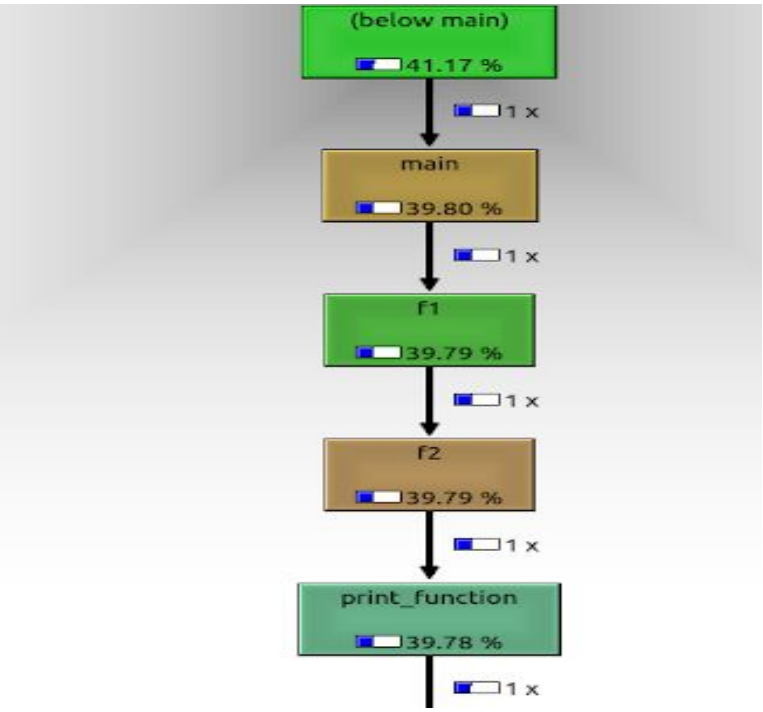


The idea is to write into a log (in our case “trace.out“) the function addresses, the address of the call and the execution time. To do so, a file needs to be open at the beginning of execution. The GCC-specific attribute “constructor” helps in defining a function that is executed before “main”. In the same way the attribute “destructor” specifies that a function must be executed when the program is going to exit.

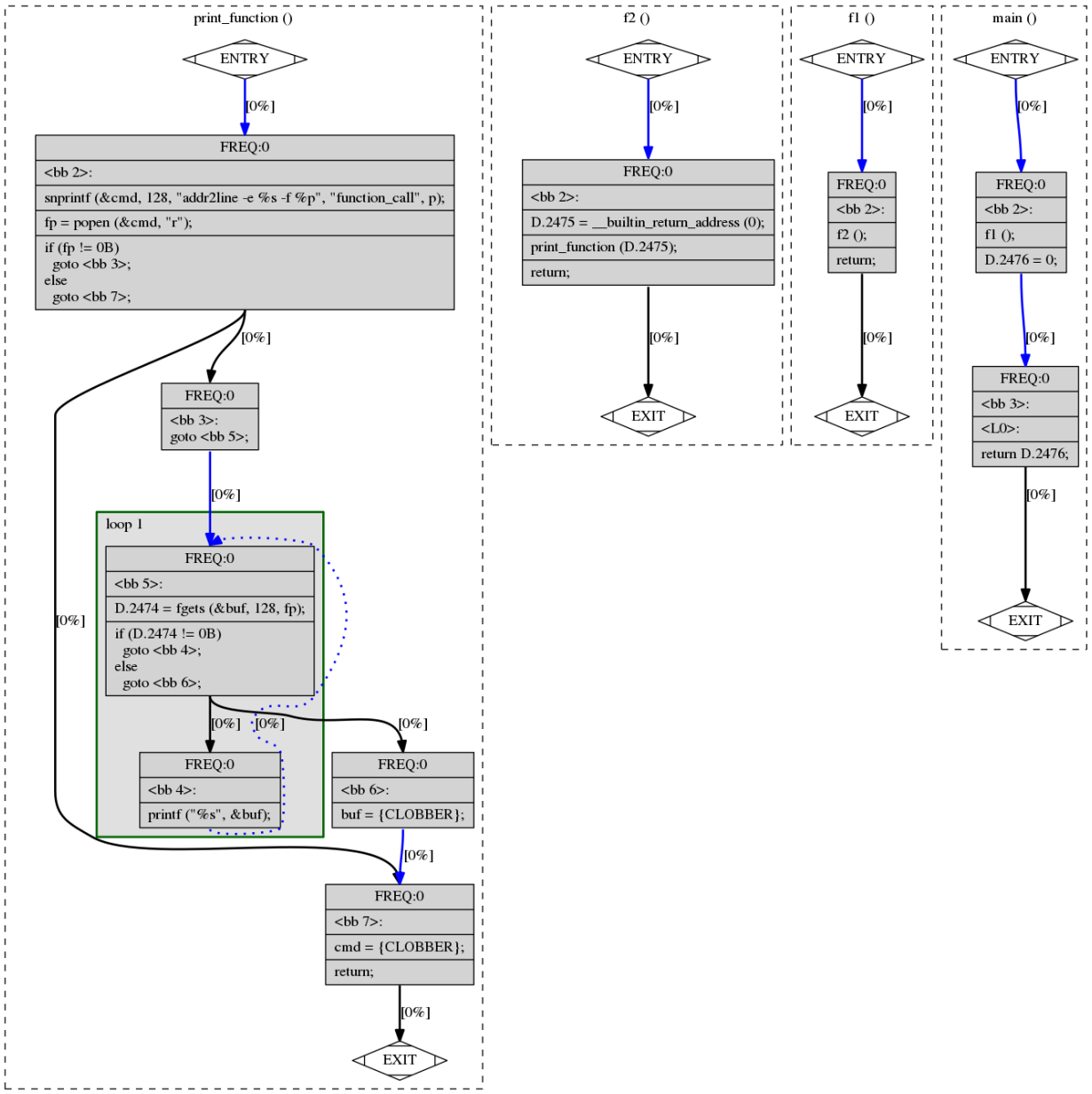
The “??” symbol indicates that addr2line has no debug information on that address: in fact it should belong to C runtime libraries that initialize the program and call the main function. In this case the execution time was very small (less than a second) but in more complex scenarios the execution time can be useful to detect where the application spends the most time.

Block Diagram :

Kcachegrind Output for the Program:



Control Flow graph of a dummy program:



Usefulness of the project

Call graphs are a basic program analysis result that can be used for human understanding of programs, or as a basis for further analyses, such as an analysis that tracks the flow of values between procedures. One simple application of call graphs is finding procedures that are never called. Software debugging is a complex task. There is always the need to collect all available information, in order to detect and understand the problem fast and to think of a proper solution. Sometimes it's more convenient to debug step-by-step, sometimes it's better to make the program run completely, and then trace the execution flow . Our project can help programmers to debug the logical errors in their program by giving them a control flow in which their program is being executed. Tools such as gbd are there in the market but we can give a graphical sense to how functions are being called. By tracking a call graph, it is also possible to detect anomalies of program execution or code injection attacks. We are also printing the memory usage used by a function call which can be useful to provide a whole bunch of optimization to our program.

References

[1] https://www.geeksforgeeks.org/compiler-design-runtime-environments/

[2] https://stackoverflow.com/questions/5945775/how-to-get-more-detailed-backtrace

[3] https://stackoverflow.com/questions/105659/how-can-one-grab-a-stack-trace-in-c

[4] https://www.linuxjournal.com/article/6391