GCC Instrumented Functions

Renato Grottesi

Abstract

Using the *-finstrument-functions* flag in GCC it is possible to inject code in the compiled object to execute a special function "*void __cyg_profile_func_enter (void *this_fn, void *call_site)*" before each other function starts and to execute another special function "*void __cyg_profile_func_exit (void *this_fn, void *call_site)*" before each other function ends.
These two simple functions are the basis to create a fairly simple framework where, each time we enter or exit a function, we record the call stack, the timestamp and the memory usage.
Such framework can be very effective on debugging applications by introducing proactive assertions that can print where the invariant was broken, or by allowing smart memory leak detection systems that can print the stack that triggered the allocation.
With an external data mining tool, the information about functions interaction collected at runtime can also be used to identify bottlenecks, to suggest source code optimizations, and to propose source code refactoring.

Table of contents

1. Introduction to GCC Instrumented Functions

Since GCC 4.6, it is possible to inject code in the compiled object to execute a special function "*void __cyg_profile_func_enter (void *this_fn, void *call_site)*" just after function entry and to execute another special function "*void __cyg_profile_func_exit (void *this_fn, void *call_site)*" before function exit.

The *this_fn* parameter will contain the address of the function and the *call_site* will contain the address of the assembly line that invoked the function. Such addresses will change at every execution since the libraries may be loaded at different memory location. However, considering the address of a specific function (or examining the /proc/#pid/maps file on linux), it is possible to extract the offset of the loaded library and calculate the base addresses of the functions. From the base address it's easy to retrieve function names or source code lines of invocation by using GNU utils like nm or objdump.

To use the two profile functions, it is necessary to compile the code using the GCC flag *-finstrument-functions* and to make sure that both *__cyg_profile_func_enter* and *__cyg_profile_func_exit* are define in the source code.

To allow the two profile functions to invoke other functions without being called recursively from them, GCC offers 3 different solutions:

1. mark the functions with the *no_instrument_function* attribute one by one in the code
2. put all the functions that should not be instrumented in files with a special name and use the flag *-finstrument-functions-exclude-file-list=file,file,...*
3. use special names for the functions that should not be instrumented and use the flag *-finstrument-functions-exclude-function-list=sym,sym...*

Another problem with the code injection comes with inline functions: while the *call_site* will be accurate, the *this_fn* parameter will change depending on where the function has been inlined. We can consider this not a problem since both nm and objdump can retrieve the name of the instrumented function, but we should keep it into account when we'll try to perform data mining on functions iterations. Either the data mining should happen with function names, or we'll have to set up an equivalency table to ensure that two inlined instances of the same function will be considered equivalent.

If function inlining is a problem or if more debug informations are needed by the two profile functions at run time, if would always be a good idea to check the GCC manual to generate a proper binary file.

2. Instrumented Functions Framework

To make a good use of the instrumented functions, it's necessary to create a proper framework around them. The framework should be thread safe and it should be able to retrieve additional data from each invocation of the two profile functions. For the purposes of this document, we are interested in enriching the *this_fn* and *call_site* parameters with the history of functions invoked before, with the call stack for that thread, with a timestamp and with the total amount of memory allocated for that thread.

2.1 Retrieving additional information

Getting the function name from the *this_fn* input parameter for meaningful messages, is not as trivial as it may seems. The solution we used in our framework is to generate an address to name dictionary at compilation time using nm, name it like the library but with the ".map" extension and place it in the *LD_LIBRARY_PATH*. The first time the *__cyg_profile_func_enter* is invoked, it will notice that the runtime dictionary is empty and it will load it from the dictionary file. The first execution of *__cyg_profile_func_enter* will also calculate the relative address at which the current library is loaded in memory to calculate the absolute addresses of the invoked functions.

The *call_site* parameter is not very useful at run time, but it's still useful to subtract the offset from all values to make them usable when parsing them offline.

Retrieving a timestamp every time a function starts or end can be a very expensive operation. Depending on the available hardware, the framework should use the less expensive to call timer with the best resolution possible. If the timestamp doesn't have an appropriate precision, it will be difficult to spot mutex contention in a timeline view, and some race conditions may show a wrong sequence of locking operations.

To retrieve the call stack for each thread, it's enough to use a LIFO structure local per thread. In our framework we used the thread local storage (TLS) to store such structure. Every time *__cyg_profile_func_enter* is invoked, a new element is add to the head of the LIFO and every time *__cyg_profile_func_exit* is called, the last element is removed from the LIFO. At every moment, we can check the stack depth and content of each thread. Each element of the stack contains the function pointer, the call site, the stack depth and the memory usage and timestamp recorded when entering that function. For our framework we restricted the LIFO to a maximum depth and keep it allocated on the stack. More sophisticated implementations may record the stack of all other threads for every LIFO entry or may allow a dynamic sized LIFO. It should be possible to dump the stack to stdout, to a file and to retrieve it and save it in a local variable that may be dump later.

After adding an element to the call stack LIFO, we found useful to save such element in a huge circular buffer containing the last 64K function invocations. Of course the LIFO element must be enriched with the thread ID before being saved to the circular buffer since it is shared by all threads. This circular buffer can then be dumped to stdout or to a file and used to check how did the system reach a particular state.

The most difficult information to retrieve is indeed the system memory usage. While it

may be possible to rely on the /proc/#pid/status information for low precision purposes, for our framework we wrapped the calls to *malloc* and *free* with our own memory tracking system. Such solution add an high precision memory tracking together with a thread safe way of retrieving the total allocated memory size.

2.2 Presenting the data

The proposed framework allows two different kinds of dumping data: call stacks and function invocation history.

Call stack dumps should print the thread id and the stack depth followed by a line for each element in the LIFO. Since it's not easy to decode the call site of a function at run time, it may be a good idea to create a call stack dump parser that can enrich the output decoding the call site.

The circular buffer containing the history of function invocations allow more complex and meaningful visualizations.
To allow an understanding of how functions interact with each other in a single thread and to speed up new starters into the code architecture, it may be a good idea to show the data as a navigable call tree. Each node on the tree can be expanded to check show an ordered list of all the functions that it invoked. Each of those function can be expanded too if it's not a leaf.
A more complex representation to show all the function interactions at once can be a call graph. Each node on the graph will represent a function and each directional arrow will connect a node to the functions it invoked at run time. With a proper layout, a graph can show very simply and without repetitions the modularity of the software architecture.
By all means, the more useful visualization remains the timeline. The advantages of the timeline is that it can plot simultaneously the status of each thread making use of the timestamp together with the graph of memory usage. As we'll see in the next chapters, a timeline analysis will highlight very complex issues like mutex locking or memory allocation policies.

Both the tree and timeline visualizations offers the opportunity to compare two different runs of the same application in a side by side view. Similar to examining a diff file of a source code file, the diff view can help the developer to spot why a patch introduced a run time bug.

3. Debugging with the Framework

In this chapter we'll examine how to use the Framework to make it easier to debug faulty applications.

3.1 Improving Assertions

A firsts trivial use of the call stack is to empower the assertion macros to print the call stack for all threads together with the error message. In this way, there will be no need to use external applications like gdb to print the stack. Together with the call stack, asserts can also

print (or save to a file) the history of the last hundreds calls: something that no other tool can offer.

A less trivial, but very powerful usage of the call stack is to save it into local object instances and print it later when something goes wrong. This couples very well with the class invariant design pattern. Every time a modifier class method is invoked, the call stack for such method is saved. When the invariant is broken, all the stacks for the modifier methods can be printed to show how the class invariant was broken.

We successfully used this method to debug a race condition that was segfaulting when a mutex was release on a thread after being destroyed from another thread. Every time the destroy method was invoked on the mutex, we saved the stack inside the mutex object. When the mutex was locked, we checked that it wasn't destroyed in an assertion and print the call stack of the last destroy. When the assert triggered after several hours of running the test, it showed exactly the race condition and we could fix it in less than ten minutes.

In a similar way, it would be possible to debug deadlock conditions by connecting gdb to a stalling application and ask each mutex to print the call stack that locked it.

The same concept applies to more complex objects like state machines. Examining the history of calls -or saving the call stacks for the last n state changes- it can be possible to examine wrong states with a quick look at the instrumentation dump.

3.2 Memory allocation debugging

Another useful usage of the call stacks is to instrument all memory allocations to detect memory leaks and double frees.

The *malloc* wrapper should associate every memory allocation in a dictionary that uses the returned pointer for addressing. When *free* receives a pointer it saves the call stack in a dictionary similar to the one used by *malloc*. If *free* finds that the dictionary already contains an entry for the same pointer, it interprets it as a double free and it prints both stacks for the first and second *free* invocations. When the application exits, the memory allocation wrapper can check that all the entries in the *malloc* dictionary have a match in the *free* dictionary and print all the call stacks for the unfreed memory allocations.

A similar approach can be used to debug memory systems that use *addref* and *deref* mechanisms to allocate and free shared memory blocks. Every *addref* should save the call stack together with the reference count. Memory leaks and *deref* calls happening after the reference count is less than or equal to zero will be easily detected.

A slightly less intuitive use of the framework for memory debugging, consists in examining the call history in a timeline visualization that plots the memory usage together with the current call stack visualized as a pyramid. We used this technique to identity the memory consumption in a complex application. Thanks to the clear timeline figure, we could notice that the memory was allocated in a thread little by little and freed all together in a different thread when a special event happened. The problem with our system was that the free event was not triggered fast enough making us use more memory than need. Without a timeline view that shows all the runtime information in a single clear picture, our team would had needed a complex and long analysis to produce the same conclusion with so detailed evidence.

3.3 Stack corruptions

Stack corruptions happen when the software write some very wrong memory locations and it is a very difficult problem to debug because neither gdb can print the call stack that triggered the corruption. The framework described in this document offers a solution for this problem. Since our call stack is stored in the current thread's local storage, it will always be possible to print it even if the actual call stack is corrupted in memory. Our implementation of the framework can actually also dump the call history from a global function, meaning that we can even retrieve the sequence of calls that corrupted the stack.
Every time *__cyg_profile_func_exit* is invoked, our framework will check that the function that is finishing is in fact the one on top of the call stack. If such condition is not satisfied, the framework reports a possible memory corruption and prints the call history.
A possible (but not yet explored) usage of the framework to verify call coherency is to verify that, for every function A that invokes a function B, there are no occurrences of function B invoking function A in the call history. While not being a bug, such conditions are very dangerous and can hide actual bugs.

3.4 Runtime diffs

It is common in complex projects to perform a binary search to find which revision introduced a bug. However the bug is often not clear by just examining the diff between the revisions. With the proposed framework is it possible to record the whole call history for the working and faulty versions and compare them side by side in a graph view. A smart graph visualizer can mark in red different subtrees of function invocations to highlight possible divergences between the two runs and point the programmer to a possible debug track.

4. Profiling with the Framework

4.1 Memory profiling

It is very difficult to find which function allocates more memory in a low level language like C. With the help of the described framework it is possible to perform a very detailed runtime memory profiling and optimize the allocation strategies.
A first form of analysis consists in grouping the dumped data by function name and calculate the sum of allocated bytes *A*, freed bytes *F*, delta *D=A-F*, number of allocations *#a* and number of frees *#f*. This very simple operation can identify all the allocators *#a>0 and #f=0*, freers *#a=0 and #f>0* and mixed functions *#a>0 and #f>0*.
With the previous grouping in place, the functions can be sorted by *A* to identify the biggest allocators in terms of requested memory and drive the attention of the developer to them.
Sorting the functions by number of allocations *#a* or frees *#f* is also very important in embedded systems where *malloc* and *free* are very expensive operations. The source code developer can probably optimize the application performance by using a memory pool to minimize the number of calls to *malloc*.

While using memory pools can reduce the amount of mallocs, it may increase the memory fragmentation together with the memory usage. A more sophisticated analysis of the memory consumption can identify functions that always allocate the same size (or a small set of different sizes) of memory. Such functions can use a proprietary memory pool that can allocate several blocks of the same size at the same time and mark each of them as available or in use. Being all the blocks of the same size, such memory pool will not suffer from any fragmentation.

4.2 Parallel code optimizations

An information that developers would like to know about their multi threaded applications is how much time does a thread spend between trying to lock a mutex and being able to complete the operation. Instead of creating a special instrumented mutex, the application developer can check the gathered runtime data in the timeline view and highlight the mutex lock/unlock functions with some special colors to have a clear overview of the problem. The timeline's user interface may also allow a button to center the graph over the longest instance of the mutex lock function in order to facilitate the search.

The data collected at runtime can be analyzed by some data mining algorithm to identify functions (or function invocation patterns) that are called several times from loops and propose them to run in parallel, either using thread pools or by using SIMD instructions.

4.3 Identify slow functions

Identifying which function to optimize can be difficult with sampling profilers like oprofile since they do not guarantee to provide the information to all the functions. On the other hand, the proposed framework will collect information for all the functions invoked at run time together with the exact call stack in which they were invoked. The biggest issue with the framework is to normalize the timeline to remove the overhead of invoking *__cyg_profile_func_enter* and *__cyg_profile_func_exit*. When the data is normalized, it can be examined on the timeline view or it can be sorted by execution time and examined from tree view.

Another feature offered by the framework is that functions can be sorted by number of times they were invoked and, on top of that, even by the longest similar stack that trigger the invocations. Sampling profilers doesn't offer such precision and can sometimes miss small functions invoked multiple times pointing instead to the function that invoked them.

Another problem that can emerge from a deeper data mining consist on the misuse of functions. The runtime should be grouped by function name and the min, max, average and standard deviation of execution time should be calculated for every function. If a function shows an high standard deviation, it means that it can be invoked in a fast or a slow way. The developer should check the stack with the minimum and maximum execution time to verify if the slow path is triggered by some wrong parameters or if it is a legal behaviour.

4.4 Suggest inline functions

Data mining of runtime execution can again be useful to identify which functions should make sense to inline in the source code.

If a function *A* is always and only invoked from a function *B*, then function *A* should be suggested

for inlining inside *B* by being marked as a *static inline* function.
Another expensive function that should be looked for it the proxy function *P* that always invoke only function *F* and nothing else. The programmer should check all those proxy functions and evaluate if they are necessary or not.
Finally, the runtime data can be used to calculate the statistical mode (range of values with the highest occurrence) of time between invocation of functions. If the statistical mode for a function is very low, it means that the function is used very often, probably from some inner loop. The developer should be aware that such functions may waste more time being invoked than actually executing something. Again, those functions should be suggested for inlining.

5. Improving code quality with the Framework

In contrast with many source code analysis tools that read the source code, we can use the framework to collect run-time data and later evaluate it under some “good quality code” metrics.

5.1 Code utilization

Compilers can strip unused functions, but sometimes it’s difficult to identify not invoked function. Probably gcov can help identify them, but it’s granularity is by lines of code, not by functions. Double crossing the function list from *nm* with the functions actually invoked at run time, it is possible to identify all the functions that were never invoked. The programmer may then think if all such functions are needed or not.

5.2 Refactoring for readability

Another difficult problem is to mark all functions that are invoked only from a thread with a special name. With the presented framework, it is possible to group all the functions invoked by thread, remove the duplicates and finally subtract the intersection to create, for each thread, a list of functions invoked only by that specific thread.
Sometimes it is difficult to read big functions that invoke many functions from within themselves. From the recorded runtime data, it is possible to check how many functions were invoked from each used function. The data can then be sorted and the source code developer can be presented with a list of functions to split into sub-functions.
In a similar but opposite use case, it should be possible to list all the functions *A* that always invoke exactly one function *B* from themselves and suggest the developer to merge functions *B* inside *A*.
FInally, it may be useful to find all functions that never invoke subfunctions and suggest them as low level or static. It would be a good idea to have some naming conventions (or to integrate the data analyzer with the source code analysis) to help the tool understanding if a function is already inside the low level layer or marked as static in the source code.
An additional analysis that can be helpful is to identify all the functions invoked just once near the beginning or just once near the end and mark them as initializers or destructors.

5.3 Refactoring to maintainability

In many applications, especially if they are state dependant, there can be dependency systems that will sort the operation on the data depending on special triggers and events. It is very difficult to analyze such pieces of software since the sequence of function invocation is not predictable. With the framework, it is possible to check the maximum size of the call stack and verify the sequence of events that triggered it. The developer can then check if it is possible to limit the stack size. Since all the function names in the deep call stack are listed, it is also possible to find patterns or recursion and suggest the developer to flatten the recursion using perhaps a loop instead.

Sometimes it is difficult to find when two functions must be called in a specific order. Analyzing the framework's output data, we can easily identify all the functions *B* that were always preceded by a function *A*. If such sequence was always invoked by the same function *C*, then *C* is already implementing the invocation sequence contract. If the sequence *A, B* is invoked by another function different from *C*, then the developer should be informed and abstract the invocation contract in a new function.

An interesting analysis of the runtime data can be an arrangeable graph view to identify how each software module interact with the other modules. The software developer can improve the interfaces between modules to improve the cohesion inside each module and to reduce the coupling between different modules.

Each function will usually invoke a set of fixed function sequences. If the set contains many different sequences of functions invocation, then the cyclomatic complexity of such function is too high. The software developer should take care of such complex functions and split them in more manageable code.

5.4 Code Reviews

Sometimes it's difficult to understand from a code review how the functions are supposed to interact at runtime. Collecting data from the unit tests of new software module can produce a very small and useful report to attach to the code review. The report can be an image showing the timeline, graph or three views of the new functions interacting with each other or with the whole system

6. Conclusions

From our experience with the framework, we learned that it can be useful to solve virtually any kind of computer software problem.

7. Related Work

Commercial applications like Coverity can offer static code analysis, but they are quite expensive. Such application can point to complex functions, but they can't suggest any source code optimizations.

GNU's gcov is a useful tool to find which lines of the source code are not executed and remove

them.

8. References

[1] GCC instrumented functions description from GCC documentation: http://gcc.gnu.org/onlinedocs/gcc-4.4.6/gcc/Code-Gen-Options.html