Visualize function calls with Graphviz

Open source software clarifies complex call structure

M. Tim Jones, Consultant Engineer, Emulex

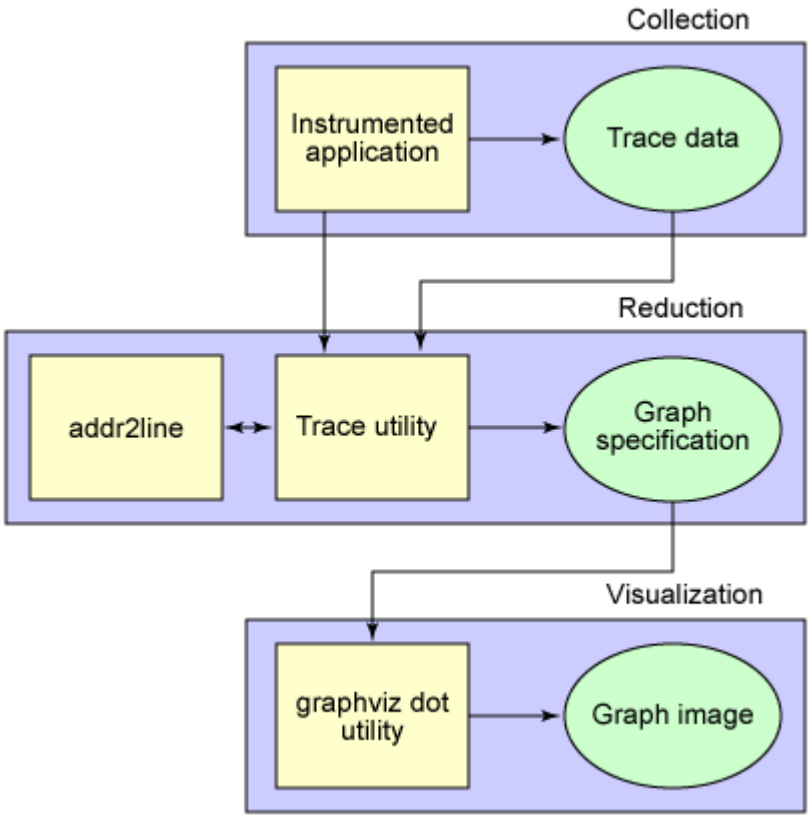
Summary: Spending the time to work through a mass of source code can reveal the function flow to you, but when function pointers are involved or the code is lengthy and convoluted, the process becomes considerably more difficult. This article shows you how to construct a dynamic graphical function call generator using open source software and a bit of custom glue code.

Date: 21 Jun 2005
Level: Intermediate
Also available in: Korean Japanese

Viewing an application's call trace in graphical form can be an educational experience. Doing so can help you understand an application's internal behavior and obtain information for program optimization. For example, by optimizing those functions that are called most often, you can get the greatest performance benefit from the least amount of effort. Additionally, a call trace can identify the maximum call depth of user functions, which you can then use to efficiently bind the memory that the call stack uses (an important consideration in embedded systems).

To capture and display a call graph, you need four elements: a GNU compiler toolchain, the Addr2line utility, custom glue code, and a tool called Graphviz. The Addr2line utility enables you to identify the function and source line number of a given address and executable image. The custom glue code is a simple tool that reduces the address trace to a graph specification. The Graphviz tool enables you to generate graph images. The entire process is shown in Figure 1.

Figure 1. Process for trace collection, reduction, and visualization



Data collection: Capturing a function call trace

To gather a function call trace, you need to identify when every function is called in your application. In the good old days, you accomplished this task by manually instrumenting each function to emit a unique symbol at its entry point and at each of its exit points. This process was tedious and prone to error, and it generally made a mess of the source code.

Fortunately, the GNU compiler toolchain (otherwise known as *gcc*) provides a means to automatically instrument the desired functions of an application. When the instrumented application is executed, the profiling data is collected. You need to provide only two special profiling functions. One function is dispatched whenever instrumented functions are called; the other is invoked when instrumented functions exit (see Listing 1). These functions are specially named so that they are identifiable to the compiler.

Listing 1. GNU profiling functions for entry and exit

```
void __cyg_profile_func_enter( void *func_address, void *call_site )
                               __attribute__ ((no_instrument_function));

void __cyg_profile_func_exit ( void *func_address, void *call_site )
                               __attribute__ ((no_instrument_function));
```

Avoid specific function instrumentation

You're probably wondering whether, if gcc is instrumenting functions, it won't also instrument the `__cyg_*` profiling functions. The gcc developers thought of this and provided a function attribute called `no_instrument_function` that can be applied to function prototypes to disable their instrumentation. Not applying this function attribute to the profiling functions results in an infinite recursive profiling loop and a lot of useless data.

When an instrumented function is called, `__cyg_profile_func_enter` is also called, passing in the address of the function called as `func_address` and the address from which the function was called as `call_site`. Conversely, when a function exits, the `__cyg_profile_func_exit` function is called, passing the function's address as `func_address` and the actual site from which the function exits as `call_site`.

Within these profiling functions, you can record the address pairs for later analysis. To request that gcc instrument all functions, every file must be compiled with the options `-finstrument-functions` and `-g` to retain debugging symbols.

So, now you can provide profiling functions to gcc that it will transparently insert into your application's function entry and exit points. But when the profiling functions are called, what do you do with the addresses that are provided? You have many options, but for the sake of simplicity, just write the addresses to a file, noting which addresses are function entry and which are exit (see Listing 2).

Note: Callsite information isn't used in Listing 2 because the information isn't necessary for this profiling application.

Listing 2. The profiling functions

```
void __cyg_profile_func_enter( void *this, void *callsite )
{
  /* Function Entry Address */
  fprintf(fp, "E%p\n", (int *)this);
}


void __cyg_profile_func_exit( void *this, void *callsite )
{
  /* Function Exit Address */
  fprintf(fp, "X%p\n", (int *)this);
}
```

Now you can collect profiling data, but where do you open and close your trace output file? So far, no changes are required to the application for profiling. So, how do you

instrument your entire application, including the `main` function, without some initialization for your profiling data output? The gcc developers thought of this, too, and provided the means for a `main` function constructor and destructor that happen to fit this need perfectly. The `constructor` function is invoked immediately prior to `main` being called; and the `destructor` function is called when your application exits.

To create your constructor and destructor, declare two functions, then apply the `constructor` and `destructor` function attributes to them. In the `constructor` function, a new trace file is opened into which the profiling address trace will be written; within the `destructor` function, the trace file is closed (see Listing 3).

Listing 3. Profiling constructor and destructor functions

```
/* Constructor and Destructor Prototypes */

void main_constructor( void )
        __attribute__ ((no_instrument_function, constructor));

void main_destructor( void )
        __attribute__ ((no_instrument_function, destructor));


/* Output trace file pointer */
static FILE *fp;

void main_constructor( void )
{
  fp = fopen( "trace.txt", "w" );
  if (fp == NULL) exit(-1);
}


void main_deconstructor( void )
{
  fclose( fp );
}
```

If the profiling functions (provided in instrument.c) are compiled and linked with the target application, which is then executed, the result is a call trace of your application written into the file *trace.txt.* The trace file resides in the same directory as the application that was invoked. As a result, you get a potentially large file filled with addresses. To make sense of all this data, you use a little-known GNU utility called Addr2line.

Resolving function addresses to function names with Addr2line

Addr2line and debuggers

The Addr2line utility provides basic symbolic debugger information, although the GNU Debugger (GDB) uses other methods internally.

The Addr2line tool (which is part of the standard GNU Binutils) is a utility that translates an instruction address and an executable image into a filename, function name, and source line number. This functionality is perfect for converting the trace addresses into something more meaningful.

To see how this process works, try a simple interactive example. (I operate directly from the shell, because it's the easiest way to demonstrate the process, as Listing 4 shows.) The sample C file (test.c) is created by `cat`-ing a simple program into it (that is, redirecting text from standard input into the file). The file is then compiled with gcc, which passes in a few special options. First, the linker is instructed (with the `-Wl` option) to generate a map file, and the compiler is instructed to generate debug symbols (`-g`). This results in the executable file *test.* Having the new executable, you can use the `grep` utility to search for `main` in the map file to find its address. Using this address and the executable image name with Addr2line, you identify the function name (`main`), the source file (/home/mtj/test/test.c), and the line number (4) within the source file.

The Addr2line utility is invoked, identifying the executable image as `test` with the `-e` option. By using the `-f` option, you tell the tool to emit the function name.

Listing 4. Interactive example of addr2line

```
$ cat >> test.c
#include <stdio.h>

int main()
{
  printf("Hello World\n");
  return 0;
}
<ctld-d>
$ gcc -Wl,-Map=test.map -g -o test test.c
$ grep main test.map
        0x08048258              __libc_start_main@@GLIBC_2.0
        0x08048258              main
$ addr2line 0x08048258 -e test -f
main
/home/mtj/test/test.c:4
$
```

Reducing the function trace data

You now have a way to collect a function address trace and also to resolve an address to a function name with the Addr2line utility. However, given the mass of trace addresses that you'll get from an instrumented application, how can you reduce the data to make sense of it? This is where some custom glue code can bridge the gap between the open source tools. The fully commented source for this utility (Pvtrace) is provided with this article, including instructions for building and using it. (See the Resources section for more information.)

Recall from Figure 1 that upon execution of the instrumented application, a trace data file called *trace.txt* is created. This human-readable file contains a list of addresses -- one per line, each with a prefix character. If the prefix is an *E*, the address is a function entry address (that is, this function was called). If the prefix is an *X*, the address is an exit address (that is, you're exiting from this function).

So, if in the trace file you have an entry address (A) followed by another entry address (B), you can infer that A called B. If an entry address (A) is followed by an exit address (A), it's understood that function (A) was called and then returned. When longer call chains are involved, it becomes more complicated to know who called whom, so a simple solution is to maintain a stack of the entry addresses. Each time an entry address is encountered in the trace file, it's pushed onto the stack. The address at the top of the stack represents the function that was last called (that is, the active function). If another entry address follows, it means that the address on the stack called the address last read from the trace file. When an exit address is encountered, the current active function has returned and the top element on the stack is discarded. This places the context back to the previous function, which is the proper flow in the call chain.

Figure 2 illustrates this concept along with the method of data reduction. As the call chain is parsed from the trace file, a connectivity matrix is built that identifies which functions call which other functions. The rows of the matrix represent the call-from address, and the columns represent the call-to address. For each call pair, the cell that intersects them is incremented (the call count). When the entire trace file has been read and parsed, the result is a compact representation of the entire call history of the application, including the call counts.

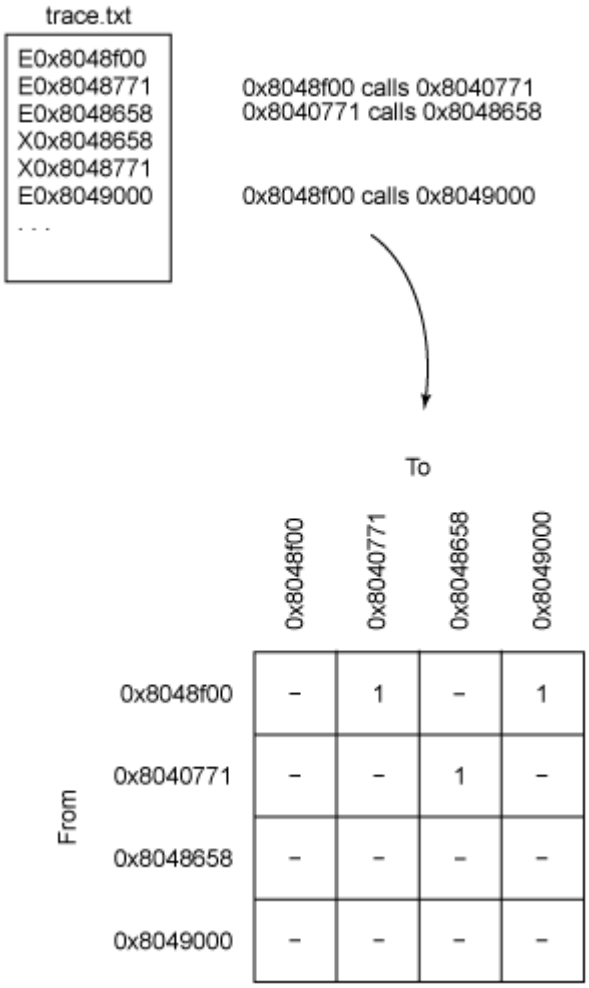
Building and installing the tools

When you've downloaded and unzipped the Pvtrace utility, simply type `make` in the subdirectory to build the Pvtrace utility. The utility can also be installed into the /usr/local/bin directory using this code:

```
$ unzip pvtrace.zip -d pvtrace
$ cd pvtrace
```

```
$ make
$ make install
```

Figure 2. Parsing and reducing the trace data to the matrix form



Now that the compact function connectivity matrix is built, it's time to build its graph. Let's dig into Graphviz to understand how a call graph is generated from the connectivity matrix.

Using Graphviz

Graphviz, or Graph Visualization, is an open source graph-visualization tool developed at AT&T. It provides several graphing possibilities, but I focus on its directed graph capabilities using the Dot language. I give you a quick overview of creating a graph with Dot and show how to turn your profiling data into a specification that Graphviz can use. (See the Resources section for information on downloading this open source package.)

Graph specifications with Dot

With the Dot language, you can specify three kinds of objects: graphs, nodes, and

edges. To understand what these objects mean, let's build an example that illustrates all three elements.

Listing 5 presents a simple directed graph consisting of three nodes in Dot notation. Line 1 declares your graph, called *G*, and its type (a digraph). The next three lines create the nodes of the graph, named *node1*, *node2*, and *node3*. Nodes are created when their names appear in the graph specification. Edges are created when two nodes are joined together by the edge operator (`->`), as shown in lines 6-8. I've also applied an optional attribute to the edge -- `label` -- that names the edge on the graph. Finally, the graph spec is completed at line 9.

Listing 5. Sample graph in Dot notation (test.dot)

```
1:  digraph G {
2:    node1;
3:    node2;
4:    node3;
5:
6:    node1 -> node2 [label="edge_1_2"];
7:    node1 -> node3 [label="edge_1_3"];
8:    node2 -> node3 [label="edge_2_3"];
9:  }
```

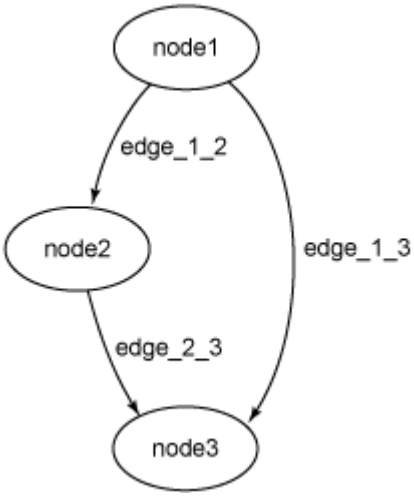
To translate this .dot file into a graph image, you use the Dot utility, which is provided in the Graphviz package. Listing 6 shows this translation.

Listing 6. Using Dot to create a JPG image

```
$ dot -Tjpg test.dot -o test.jpg
$
```

In this code, I instructed Dot to use my test.dot graph specification and generate a JPG image in the file test.jpg. The resulting image is shown in Figure 3. I used the JPG format, but the Dot tool supports other image formats as well, including GIF, PNG, and postscript.

Figure 3. Sample graph created by Dot



The Dot language supports several other options, including shapes, colors, and a large number of attributes. But for what I want to accomplish, this option works fine.

Bringing the pieces together

Now that you've seen all the pieces of the process, a single example to demonstrate the process will bring it all together. At this point, you should have extracted and installed the Pvtrace utility. You should also have copied the instrument.c file into your working source directory.

In this example, I have a source file called *test.c* that I plan to instrument. Listing 7 shows the entire process. At line 3, I build (compile and link) the application with the instrumentation source (instrument.c). I execute `test` at line 4, then use the `ls` utility to verify that the trace.txt file was generated. At line 8, I invoke the Pvtrace utility and provide the image file as its only argument. The image name is necessary so that Addr2line (invoked from within Pvtrace) can access the debugging information in the image. At line 9, I perform another `ls` to ensure that Pvtrace generated a graph.dot file. Finally, at line 12, I use Dot to convert this graph specification into a JPG graph image.

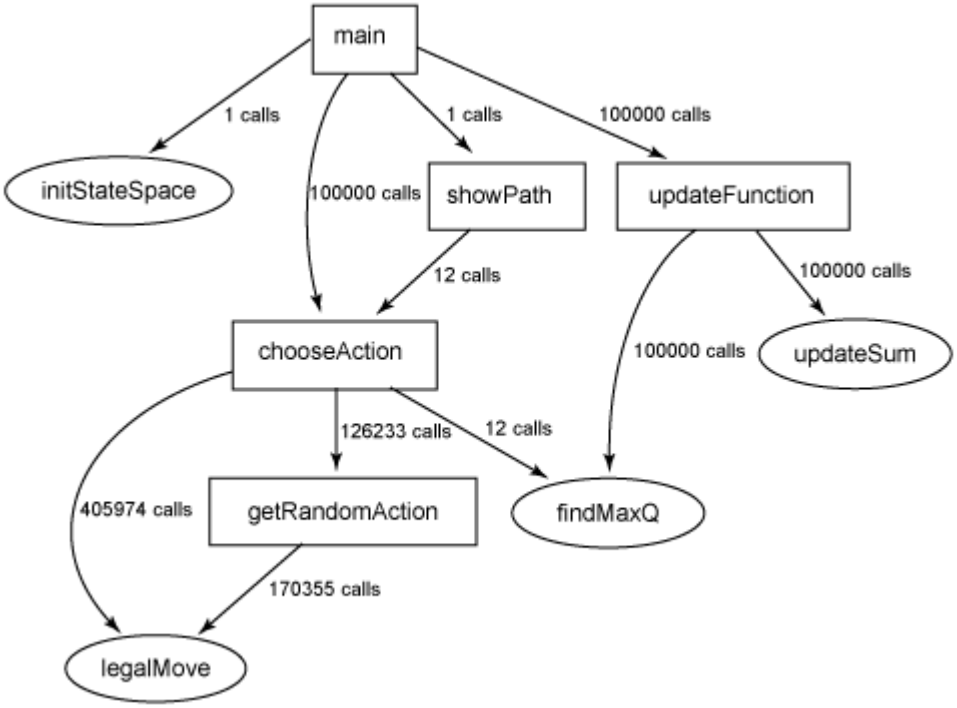
Listing 7. The entire process of creating a call trace graph

```
 1:  $ ls
 2:  instrument.c    test.c
 3:  $ gcc -g -finstrument-functions test.c instrument.c -o test
 4:  $ ./test
 5:  $ ls
 6:  instrument.c     test.c
 7:  test             trace.txt
 8:  $ pvtrace test
 9:  $ ls
10:  graph.dot        test           trace.txt
11:  instrument.c     test.c
```

```
12:  $ dot -Tjpg graph.dot -o graph.jpg
13:  $ ls
14:  graph.dot        instrument.c   test.c
15:  graph.jpg        test           trace.txt
16:  $
```

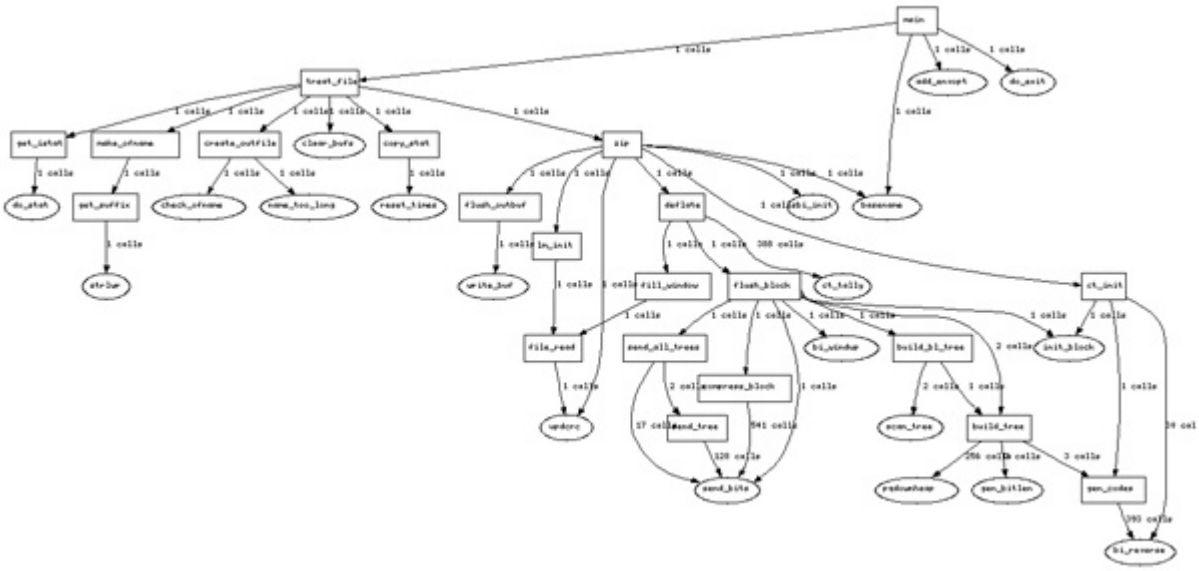
A sample output of the process is shown in Figure 4. This sample graph is from a simple reinforcement learning application that uses Q learning.

Figure 4. Sample program trace result



You can also use this method to view much larger programs. One final example I'd like to present is an instrumented Gzip utility. I've simply added instrument.c to Gzip's dependencies in its Makefile, built it, and then used Gzip to generate a trace file. This image is too large to show much detail, but the graph represents Gzip in the process of compressing a small file.

Figure 5. Gzip trace result



Summary

With open source software and a small amount of glue code, you can develop interesting and useful projects in a short amount of time. By using several GNU compiler extensions for application profiling, the Addr2line utility for address translation, and the Graphviz program for graph visualization, you get a program that profiles an application and then presents a directed graph illustrating the call chains. Viewing a program's call chain graphically is a powerful way to understand that program's internal behavior. This knowledge can be useful for debugging and for optimizing an application through an understanding of the call chains and their respective frequencies.

Resources

- Download the instrumentation and Pvtrace source code described in this article.
- See the latest GNU Compiler Collection (GCC) documentation.
- Visit the `Addr2line` main page at gnu.org to learn more about the Addr2line utility.
- Learn about the other GNU Binutils at gcc.org.
- Explore other possibilities at the Graphviz Web site.
- The manual for the Dot utility outlines the tool's other capabilities.
- Discover the Gzip utility and its capabilities.

- Read *GNU/Linux Application Programming* by M. Tim Jones to learn more about GNU/Linux and open source tools.
- "Mastering Linux debugging techniques" (developerWorks, August 2002) outlines techniques for debugging in the context of four problem scenarios.
- "Kernel debugging with Kprobes" (developerWorks, August 2004) shows how to use Kprobes to dynamically insert printk's to assist in Linux kernel debugging.
- Find more resources for Linux developers in the developerWorks Linux zone.
- Get involved in the developerWorks community by participating in developerWorks blogs.
- Browse for books on these and other technical topics.
- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Innovate your next Linux development project with IBM trial software, available for download directly from developerWorks.

About the author

M. Tim Jones is an embedded software engineer and the author of *GNU/Linux Application Programming*, *AI Application Programming*, and *BSD Sockets Programming from a Multilanguage Perspective*. His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a senior principal engineer at Emulex Corp.

developerWorks: Sign in

IBM ID:

Need an IBM ID?
Forgot your IBM ID?

Password:

Forgot your password?
Change your password

Keep me signed in.

By clicking **Submit**, you agree to the developerWorks terms of use.

Submit Cancel

The first time you sign into developerWorks, a profile is created for you. **Select information in your profile (name, country/region, and company) is displayed to the public and will accompany any content you post**. You may update your IBM account at any time.

All information submitted is secure.

Close [x]

Choose your display name

The first time you sign in to developerWorks, a profile is created for you, so you need to choose a display name. Your display name accompanies the content you post on developerWorks.

Please choose a display name between 3-31 characters. Your display name must be unique in the developerWorks community and should not be your email address for privacy reasons.

Display name: (Must be between 3 – 31 characters.)

By clicking **Submit**, you agree to the developerWorks terms of use.

Submit Cancel

All information submitted is secure.

1 star1 star

2 stars2 stars

3 stars3 stars

4 stars4 stars

5 stars5 stars

Submit

Add comment:

Sign in or register to leave a comment.

Note: HTML elements are not supported within comments.

Notify me when a comment is added1000 characters left

Post

There is a problem in retrieving the comments. Please refresh the page later.