AP GCC - CGA tool chain on unix systems

Torsten Becker, Frederik Rudeck, Robert Timm

2009-08-01

Abstract

The CGA - Call Graph Analyzer - system developed at the HPI was indented for Windows based systems only. We ported this tool chain to Mac OS X and Linux systems.

TEMPLATE

```
Inhaltsvorlage zur Dokumentation - Seminar Softwarevisualisierung SoSe09
Arbeitspaket: Linux/MacOS-Portierung von CGA
Portierung von CGA nach Linux/MacOS
Hindernisse bei der Portierung
todo
Coding Richtlinien zur zukünftigen Vermeidung von Cross-Plattform Problemen
Todo
Cross-Plattform Buildsystem CMake
Nutzung von CMake unter Windows, Linux und MacOS
todo
CGA Coding Richtlinien zur Nutzung von CMake
Todo
Portierung des Windows-basierten Faktextraktionsmechanismus Callmon nach Linux/MacOS
Analysierbare Softwaresysteme - Notwendige Voraussetzungen
Mit welcher GCC Version muss das zu analysierende System gebaut sein?
Gibt es bestimmte Systemlibs, gegen die das zu analysierende System gelinkt werden muss?
Sind Inkompatibilitäten zu erwarten mit irgendwelchen Systemlibs?
...
Todo
Workflow der Callmon Toolchain
Todo
Einbindung von 3rd Party Tools
Welche Datenquelle für Debug-Informationen wird verwendet? Gibt es Alternativen?
Warum die eine gewählt?
Wie werden die Funktionseinsprungsaddressen und Funktionsausstiegsadressen bestimmt?
Gibt es Alternativen? Welche Unterschiede gibt es bei Linux und Mac?; Todo
Implementierung der Callmon Bibliothek unter Linux/MacOS
Wie ist der Callmon-Kern nach Linux/MacOS portiert worden?
Wie wurde der Start/Stopp-Mechanismus übertragen?
Wie der Mutex-freie Ereignissammelmechenismus und wie die Ereignis-Serialisierung? ...
Todo
Tutorial: Kontrollfluss-Analyse von Audacity
Charakterisierung des Softwaresystems (LOC, etc.)
Todo
Exemplarisches Vorgehen bei der Faktextraktion (und anschließenden Visualisierung)
Todo
```

Contents

1 Introduction

2 Results

This section provides a short overview on what we achieved during in project.

2.1 CMake build enviroment

We switched the build environment of CGA to CMake. This enables the same build configuration to manage the build process on several platforms for several compilers by generating makefiles or project files for lots of IDEs.
We tested this system for Visual C++ on Windows and for Makefiles on Linux and Mac OS X. It is very likely that his will also work without any tweaks for KDevelop, XCode and many many more.

2.2 Callmon runtime library

We implemented a callmon runtime library for applications and libraries build with GCC. This library features a lock free path from call event to log file (even in multithreaded environments using atomic operations), starting and stopping of the logging process using file system event APIs and asynchronous IO to handle writing of log files as efficient as possible. Finally we benchmarked IO throughput of up to 70MB per second (MacBook Pro Unibody running ioQuake3).

2.3 Patch and Patchclean

We implemented tools to patch executables and libraries on Linux and Mac OS X. These tools store patch information in a very efficient way to binary patch files, which consume very little disk space.

2.4 Metacreator

To port metacreator to Linux and Mac OS X we just implemented the DiaInterface. So we were able to port the whole tool with very little code changes.
We added a class which gets instantiated from the DiaFactory and then interfaces with GDB to obtain debugging information like file names, line numbers of functions and call sites. Caching allows to reduce the communication with GDB to a minimum.

2.5 Cross platform fixes for the CGA Toolbar

Besides some fixes for cross platform compatibility, the CGA Toolbar is exactly the same and can be used as usual for starting and stopping the logging process.

2.6 Cross platform fixes for the CGA application

Finally we fixed several issues in the CGA application to allow the operation on Windows and Unix platforms.

3 Challenges

This section describes challenges we had to face while porting the CGA framework to Mac OS X and Linux.

3.1 Complexity

The first challenge was the complexity of CGA. The main application itself contains more than 70.000 lines of code. The whole distribution consists of about 170.000 lines of code.
Furthermore, analyzing an application using CGA requires several distinct steps, like adjusting the build process of the application that is getting analyzed, patching the applications binary and post processing the data collected while running the application. Porting this tool chain to another operating system requieres a deep understanding of all those processes on one hand, and on the other hand a good idea how to realize all those details on the target platform.

3.2 Platform specifics

The whole CGA tool chain relies on lots of platform specific mechanisms like binary patching and collecting of information from the dynamic linker. A big challenge was to find ways to get all the information needed on both target platforms.
Parts of our implementation rely on GCC and GDB, which are both available on Linux and Mac OS X. They provided us with a good point of abstraction to hide platform specific details. But even with those tools, certain thing behave differently on both platforms. Writing into a binary with GDB does not work on Mac OS X, but does work on Linux. Function call addresses reported by the GCC instrument function mechanism may be wrong on Linux. Just to name two examples.
DllMain is a mechanism which enables a developer of a dynamic library to react on events like getting loaded, getting unloaded, new threads attached and similar. This mechanism is not available on Mac OS X's .dylib system nor on Linux's .so system. So we had to make excessive use of lazy initialization, thread specific storage and similar constructs.

3.3 Compiler specifics

The main instrumentation mechanism if based on a feature provided by the compiler. The Microsoft Visual C++ can insert calls to instrumentation functions right after a function was called and right before a function returns. The mechanism in general is the same using GCC, but the differences appear when it comes to details.
On Visual C++, it is possible to compile a function *naked*, which removes functions prolog and epilog and lets the programmer implement them himself. This feature enables the function to have a certain view on the stack, because the implementation itself is responsable for creating the stack frame, adjusting stack pointers and so on. So as a *naked* function starts, it has the same view on the stack as the function which called it. This is great for the implementation of the instrumentation functions. GCC as well does provide this feature, but, it is not supported on x86 platforms. So we had to work around this situation.
Some data types, like hash_map, which are not part of the C++ Standard, have different names and reside in different namespaces on different compilers.

3.4 IDE specifics

While introducing the CMake based build system in CGA, we found ourselves in front of a complex and highly platform specific Visual Studio solution with lots of inter project dependencies. It contained lots of custom build steps, like Qt preprocessing steps (uic and moc) and post build steps to get, for example, unit testing data in place.

Furthermore the solution was a grown structure, so several obsolete code files still exist in the source tree, but are excluded from the build process. Includes defined using the Visual Studio project settings were missing in the source files which actually needed them. Just to name a few pitfalls.

3.5 Backport to Windows

Porting CGA to Linux and Mac OS X was only the first step. After we applied all the changes to the source tree and integrated CMake as a build system, be had to backport to Windows and build CGA on Windows using the new CMake build system.

3.6 Merging

Due to the requirement, that all the changes made during the seminar should be able to easily get applied to the trunk of CGA, we had to constantly merge changes from the seminars main branch to our own one. Those changes again brought incompatibilities with gcc and even our unix fact extraction port which partly relies on code from the Windows fact extraction mechanisms.
When all the other project branches merged their changes into the seminars main branch, we as well pulled that changes into our branch to ensure Linux and Mac OS X compatibility of those projects.

3.7 Qt did a great job

In general, we have to say, that Qt did a great job. Without the platform independence of not only all the GUI code, it would not have been possible to port CGA in such a short time periode.

4 Requirements

This section describes some version requirements for the tools we are using.

4.1 Summary

- a patched version of VRS (see below for details)
- GCC tested on 4.2, 4.3 (need minimum version 4.2)
- GDB tested on 6.3, 6.8
- BINUTILS tested on 2.19.1, XCode 3.1.3

4.2 Patched version of VRS

In order to get CGA running on Mac OS X and Linux, we needed to apply three patches to VRS. One of them is necessary to get VRS to compile, the second one is a header only fix for VRS, without it, CGA would not compile and the third one is a workaround for a crash in VRS, happening when using CGA. The three patches can be downloaded at GitHub:

- http://github.com/torsten/ap_gcc/blob/b4847070f55198fcee57102c959adc0c4c794b5c/vrs-trunk-r6691-gcc-build-fix.patch
- http://github.com/torsten/ap_gcc/blob/b4847070f55198fcee57102c959adc0c4c794b5c/vrs-trunk-r6691-linux-cga-context-foo-crash-workaround.patch
- http://github.com/torsten/ap_gcc/blob/b4847070f55198fcee57102c959adc0c4c794b5c/vrs-trunk-r6691-mac-qt4-debug-only.patch

4.3 GCC Version

We need a GCC version, which is greater than or equal to version 4.2 because we are using some functionality which appeared in GCC version 4.2 for the first time.
For example with the function *__sync_bool_compare_and_swap* GCC provides us with a cross platform way to test and set a variable in an atomic way. This is used several times in the lock free callmon implementation and therefor is essential.
We as well tested our code on GCC 4.3 without any problems.

4.4 GDB Version

Our implementation of the metacreator was tested on GDB version 6.3 (on Apple Mac OS X) and 6.8 (on Ubuntu Debian Linux).
We do not depend on any functionality which was introduced in version 6.3, so our implementation may also run on older and/or newer versions of GDB. But since we are using GDB in a terminal way (writing commands to it and reading its output), we highly depend on the formatting of GDBs output. Even between version 6.3 and 6.8 there were several small differences like additional line breaks in GDBs output.
But extending our implementation to handle more versions of GDB is as easy as fixing the regular expressions parsing the GDB output.

4.5 Binutils

On Linux we are using *objdump* and *nm* from the binutils distribution. All our code was tested with version 2.19.1 of these tools. On Mac OS X we a using *otool* and *nm* as provided by Apple bundled with XCode version 3.1.3.
Like in the GDB case we are parsing the output using regular expressions, so changes in the output format of these tools are likely to break our parsing, but again, only some regular expression need to be adjusted.

5 CMake Build System

keine special coding richtlinien

5.1 Qt specific build steps

wo landen die uic generierten dateien

5.2 on Windows

[[CMake GUI - grouped view]]

5.3 on Linux

5.4 on Mac OS X

6 Unix fact extraction mechanism in detail

In this section we want to explain the unix fact extraction mechanism in more detail. Especially we want to explain how we implemented the callmon library, the executable patching mechanism and metacreator.

6.1 Overview

First of all we want to give a quick overview about the whole workflow of the fact extraction tool chain. These are the steps if you want to instrument your application and visualize the trace in CGA.

1. Building the application with special compiler flags and linking the callmon.dll,dylib,so.
2. Patching the executable results in an executable with just NOPs and a patch file (patch clean).
3. Patching the executable with the help of an include/exclude file so that we just have specific logging enabled (patch).
4. Executing the application.
5. Start logging with the cga toolbar creates a pre modinfo file.
6. During the logging callmon writes cmlog files for every thread.
7. Stop logging with the cga toolbar creates a post modinfo file.
8. The metacreator takes the cmlog and modinfo files and creates a .callmon file
9. Import and visualize the trace with CGA.

For more detailed information about the process have a look at section 7.

6.2 Callmon

The central element of the tracing mechanism is callmon. Callmon is a static library, which you have to link against your application you want to instrument. But the most important part is the gcc parameter -finstrument-functions. We will explain this later in more detail. To get an overview about the classes which are involved in the tracing mechanism have a look at figure 1.

The *StartExitHandler* helps to set crash handlers for the most common signals (SIGINT, SIGILL, SIGSEGV, SIGFPE, SIGTERM, SIGABRT, SIGBUS, SIGHUP)[1] and an exit handler right after startup (before main). We ensure that the constructor of the *StartExitHandler* is executed before the main routine of the application with the help of a static variable. Static variables are initialized before the first call of the application main routine. The *exithandler()* function deletes the *Coordinator* instance. If the application crashes the crash handler gets executed which processes just a normal cleanup.

As mentioned early one of the key elements is the gcc parameter -finstrument-functions. **[[cyg func enter]]** The callmon class ... **[[explain]]**

```
__cyg_profile_func_enter, enter procedure, __cyg_profile_func_exit,
exit procedure, dereferenceLinuxJumpTable
```

callinfostack, shadowstack
thread local storage Callmon singleton pro thread (lazy initialization, is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.)
Event explain structure: function address, return address, object address, time, flag (entry or exit)

[1]http://en.wikipedia.org/wiki/Signal_(computing)

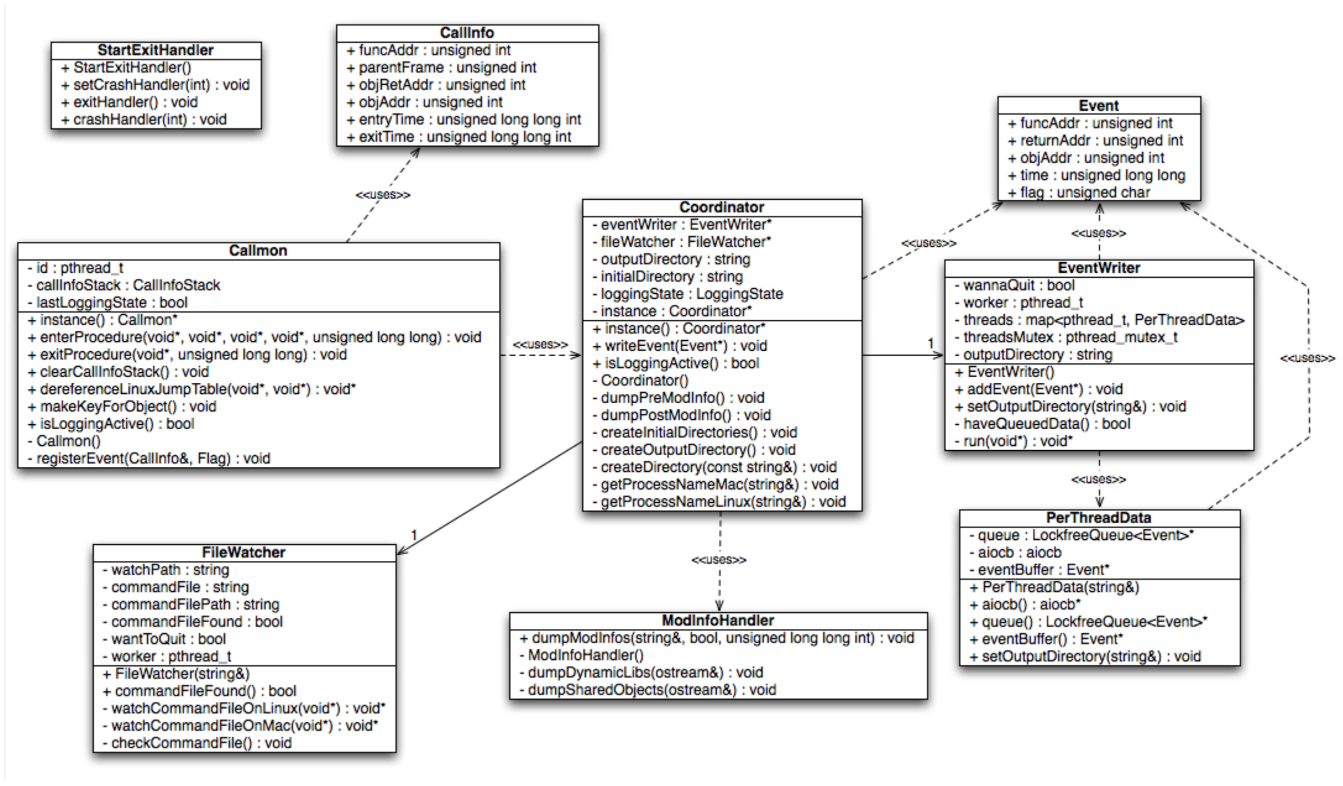
StartExitHandler
+ StartExitHandler()
+ setCrashHandler(int) : void
+ exitHandler() : void
+ crashHandler(int) : void
CallInfo
+ funcAddr : unsigned int
+ parentFrame : unsigned int
+ objRetAddr : unsigned int
+ objAddr : unsigned int
+ entryTime : unsigned long long int
+ exitTime : unsigned long long int
<<uses>>
Callmon
- id : pthread_t
- callInfoStack : CallInfoStack
- lastLoggingState : bool
+ instance() : Callmon*
+ enterProcedure(void*, void*, void*, void*, unsigned long long) : void
+ exitProcedure(void*, unsigned long long) : void
+ clearCallInfoStack() : void
+ dereferenceLinuxJumpTable(void*, void*) : void*
+ makeKeyForObject() : void
+ isLoggingActive() : bool
- Callmon()
- registerEvent(CallInfo&, Flag) : void
<<uses>>
Coordinator
- eventWriter : EventWriter*
- fileWatcher : FileWatcher*
- outputDirectory : string
- initialDirectory : string
- loggingState : LoggingState
- instance : Coordinator*
+ instance() : Coordinator*
+ writeEvent(Event*) : void
+ isLoggingActive() : bool
- Coordinator()
- dumpPreModInfo() : void
- dumpPostModInfo() : void
- createInitialDirectories() : void
- createOutputDirectory() : void
- createDirectory(const string&) : void
- getProcessNameMac(string&) : void
- getProcessNameLinux(string&) : void
Event
+ funcAddr : unsigned int
+ returnAddr : unsigned int
+ objAddr : unsigned int
+ time : unsigned long long
+ flag : unsigned char
<<uses>>
<<uses>>
EventWriter
- wannaQuit : bool
- worker : pthread_t
- threads : map<pthread_t, PerThreadData>
- threadsMutex : pthread_mutex_t
- outputDirectory : string
+ EventWriter()
+ addEvent(Event*) : void
+ setOutputDirectory(string&) : void
- haveQueuedData() : bool
- run(void*) : void*
1
<<uses>>
<<uses>>
PerThreadData
- queue : LockfreeQueue<Event>*
- aiocb : aiocb
- eventBuffer : Event*
+ PerThreadData(string&)
+ aiocb() : aiocb*
+ queue() : LockfreeQueue<Event>*
+ eventBuffer() : Event*
+ setOutputDirectory(string&) : void
1
FileWatcher
- watchPath : string
- commandFile : string
- commandFilePath : string
- commandFileFound : bool
- wantToQuit : bool
- worker : pthread_t
+ FileWatcher(string&)
+ commandFileFound() : bool
- watchCommandFileOnLinux(void*) : void*
- watchCommandFileOnMac(void*) : void*
- checkCommandFile() : void
<<uses>>
ModInfoHandler
+ dumpModInfos(string&, bool, unsigned long long int) : void
- ModInfoHandler()
- dumpDynamicLibs(ostream&) : void
- dumpSharedObjects(ostream&) : void

Figure 1: Class diagram of the tracing mechanism

6.2.1 Coordinator

singleton
Coordinator - create initial directories (get proc names, linux and mac), modinfohandler, isLoggingActive, output directory

6.2.2 Filesystem events

linux filesystem watching
mac file system watching
cga toolbar

6.2.3 Lockfree event to disk

6.2.4 Asynchronous IO

6.3 Patch and Patchclean

6.3.1 ELF patching

6.3.2 MachO patching

6.4 Metacreator

6.4.1 Line number caching

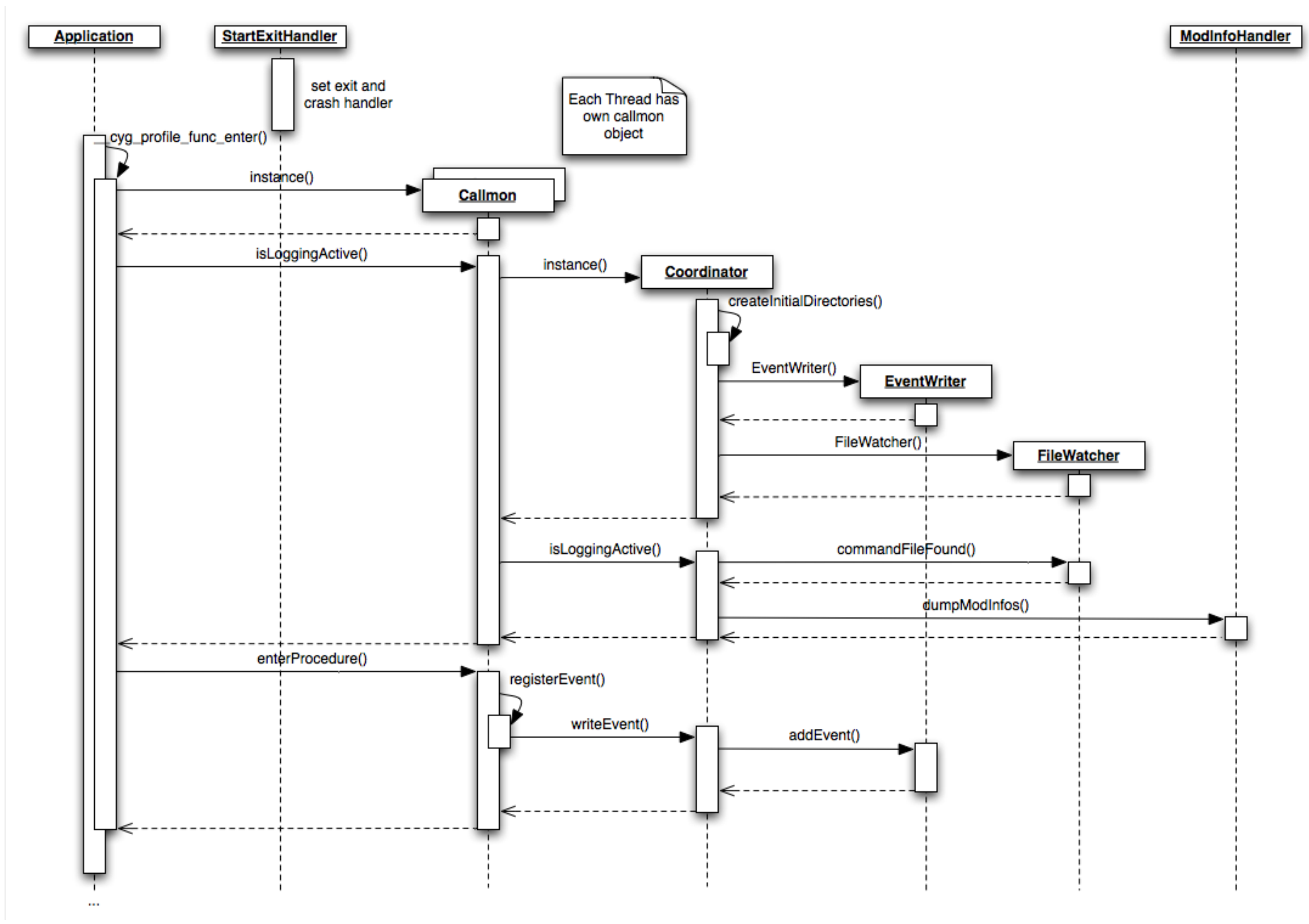

Figure 2: Sequence diagram of the function enter procedure

7 Tutorial

This section shows how to profile a project step by step. In general, the process is quite similar to the process needed to profile a project with the Windows version of callmon. Big differences only appear in the build configuration due to differences in the compiler switches needed by GCC.

7.1 Preparing the build process

How to prepare the build process.

7.1.1 Parameters to GCC

This is a list of parameters needed by GCC when profiling with unix fact extraction.

Compile with -g Enable debug information.
With this option enabled, GCC builds debug information into the resulting binary. This is needed by metacreator to resolve source file name, line numbers and other valuable debugging information.

Compile with -finstrument-functions Enable instrumentation.
This GCC parameter is the key. With this option enabled, GCC inserts calls to instrumentation functions right after a function was called and right before a functions return. If you do not provide this option in the compilation process, no calls will be made to the unix callmon lib, and therefor no profiling information can be retrieved.

Compile with -fno-inline Disable inlining of functions.
This disables the inlining of functions which is done by the compiler automatically to optimize execution speed by eliminating function call overhead. Since we are profiling on a function execution level, we cannot profile inlined function, so all the functions inlined by the compiler cannot appear in the call graph. To be sure this cannot happen, use -fno-inline as a GCC option. You might skip this if you want. You still might get good profiling results for the calls you are interested in, but you have been warned! Take care!

Link unixcallmon library as the last library Ensure the right profiling functions are used.
It might happen, that other libraries as well provide the profiling functions that unix callmon lib provides (glib is an example for that). If this happens, it is important to ensure that the versions of the unix callmon lib are the ones linked in. This is done by adding the -lunixcallmon_lib parameter as the last one.

On Linux, link with -Wl,-Bsymbolic Make function calls patchable.
Without this option, symbolic function call information is removed from the resulting binary on Linux. This prevents patch and patch_clean from finding the call locations and makes it impossible to remove those calls from the binary.

Build with absolute path to source Ensure CGA can find source files.
To ensure that CGA can find the source files for the source code viewer, you need to provide GCC with the absolute path to the source file while compiling.

7.1.2 Putting it all together

Íf the project you want to analyze is build using make, you might want to add the following to the projects Makefile:

```
CFLAG   += -g -finstrument-functions -fno-inline
LDFLAGS += -Wl,-Bsymbolic -L/path/to/callmonlib -lunixcallmon_lib
```

7.2 Building the application

With all the above mentioned set up, you build your project as usual, eg. by typing *make*.

7.3 Patching the executable

As soon as the build process has finished, you end up with an executable or library which has all the profiling mechanisms build in. At this point, every single function which was just compiled by GCC is now enriched by profiling logic. Since this may be a lot, you might want to exclude several functions or groups of functions from the profiling process. This is done by patching the binary. Technically, the calls to the instrumentation functions get overwritten by NOP operations, so almost no overhead is involved in calling functions removed from the profiling process.

7.3.1 Patch clean

The first thing to do is call patch_clean on the binary like this:

```
$ patch_clean myBinary myBinary.patch
```

The first parameter specifies the binary to patch. The second parameter specifies a patch file name. In this patch file, patch_clean will write out all the locations from which profiling calls were removed along with the opcodes removed that realized the call. This information in needed by patch to re-include the call opcodes for certain functions in the next step.

7.3.2 Patch

At this point, profiling logic is re-added to the functions of interest. This is done by specifying two groups of function name patterns in a pattern file. This is exactly the same like for Windows callmon.
Provide a list of patterns in the include section, as well a list of patterns in the exclude section. Then, call patch like this:

```
$ patch myBinary myBinary.patch myPatternsFile.txt
```

You may leave the patterns file parameter empty to re-include all functions in the profiling process. Like this:

```
$ patch myBinary myBinary.patch
```

7.4 Using CGA Toolbar

The CGA Toolbar can be used as usual. It needs to environment variable CALLMON_HOME set sup. This variable has to contain the path, where the log files are created. So lets say, you have a directory structure like this:

```
<working directory>/
|
|- myBinary
|- myBinary.patch
|- myPatternsFile.txt
```

Create a directory where the CGA Toolbar can operate on, like this:

```
$ mkdir -p logs/myBinary
```

You end up with a directory structure like this:

```
<working directory>/
|
|- logs/
|  |
|  |- myBinary/
|     |
|     |
|
|- myBinary
|- myBinary.patch
|- myPatternsFile.txt
```

Now, fire up the CGA Toolbar like this:

```
$ CALLMON_HOME="<working directory>" cgatoolbar
```

The string *myBinary* should now show up in the drop down menu in the CGA Toolbar. If you now hit the start button, CGA Toolbar will create a file called callmon.cmd:

```
<working directory>/
|
|- logs/
|  |
|  |- myBinary/
|     |
|     |- callmon.cmd
|
|- myBinary
|- myBinary.patch
|- myPatternsFile.txt
```

This tells callmon to log function calls. Hitting the stop button will remove this file. You are now ready to run you application and record profiling information.

7.5 Running the application

Run the application as usual for example like this:

```
$ ./myBinary -someParameter=someValue
```

If you now press start and stop on the CGA Toolbar, new traces will be generated. Each trace will reside in the logs/myBinary/ directory. Each trace will be put into its own directory depending on the date and time the logging started at. So after you created several traces, you might end up with a directory structure like this:

```
<working directory>/
|
|- logs/
|  |
|  |- myBinary/
|     |
|     |- callmon.cmd
|     |- 090229_143523
|     |- 090229_143542
|     |- 090229_143559
|     |- 090229_143614
|
|- myBinary
|- myBinary.patch
|- myPatternsFile.txt
```

Each trace directory now contains .cmlog files, each of them representing the events that occured in one thread and .modinfo files, that contain the dynamic library state when logging started and as well when logging ended. So the directory for one trace may look like this:

```
090229_143542/
|
|-profile_4243_b8bfa41d.cmlog
```

```
|-profile_4243_b8bfd411.cmlog
|-profile_4243_b1ad00d2.cmlog
|-profile_4243_pre.modinfo
|-profile_4243_post.modinfo
```

The first number in the .cmlog filename describes the process identifier, the second number in hexadecimal describes the thread identifier. The .modinfo filenames as well contain the process identifier.

7.6 Using Metacreator

The next step is to enrich the collected information by running metacreator. Therefor you simple fire up metacreator and provide it with a traces directory as parameter, like this:

```
$ metacreator ./logs/myBinary/090229_143542
```

Depending on the amount of events you collected, metacreator will take some time now. For all the logged calls, metacreator will now resolve debugging information from the binary. Once finished, a new .callmon file was created in the traces directory. So it should look like this now:

```
090229_143542/
|
|-profile_4243_b8bfa41d.cmlog
|-profile_4243_b8bfd411.cmlog
|-profile_4243_b1ad00d2.cmlog
|-profile_4243_pre.modinfo
|-profile_4243_post.modinfo
|-profile_4243.callmon
```

All the preparations are done now. You can now start up CGA and load the trace.

7.7 Loading the trace(s) into CGA

Start the CGA executable, create a new project. Then, select manage traces, click the add button, select your .callmon file and let CGA import the trace.

8 Guideline - Code that builds on GCC and Visual C++

This section contains a list of the most common problems we found in the code of CGA. As well we provide a solution to these problems, which make the code compile and run on Visual C++ and GCC.

8.1 Paths

To be valid on Windows and Unix platforms, a path **must not** contain any backslashes as separators. The only valid path separator for both platforms is the slash symbol /. So a valid path looks like this:

```
"this/is/a/valid/path"
```

This applies to paths needed by the programs logic, for like opening files, and as well for paths used in the build process, for example includes.

8.2 Const correctness

There were several parts of the code which made use of the keyword *const* in an invalid way. We still do not know, how this was able to compile on the Visual C++ compiler.
When declaring something *const*, it **must not** be changed or given to a function that expects a non const, because this function could potentially change it. An example:

```
#include <QtCore/QString>

void changeMyString(QString& p_string) {
  p_string.replace("foo", "bar");
}

QString changeMyConstString(const QString& p_string) {
  QString result = p_string;
  result.replace("foo", "bar");
  return result;
}

int main() {
  const QString myString("foobarlalala");

  // THIS IS INVALID and will cause a COMPILER ERROR
  // error: invalid initialization of reference
  //        of type ‘QString&’ from expression of type ‘const QString’
  changeMyString(myString);

  // this is ok
  QString newString = changeMyConstString(myString);

  return 0;
}
```

8.3 Returning references to temporaries

When returning references to objects, it is very important to ensure where the object is living. If a function returns a reference to one of their local variables, this is very dangerous. An example:

```
#include <QtCore/QString>

QString& giveMeAString() {
  QString localString("foobarlalala");

  // throws a compiler warning
  // warning: reference to local variable ‘localString’ returned
  return localString;
}

int main() {
  // this is dangerous because the scope of the variable localString was
  // just left as the function returns. So myStringRef is invalid.
  QString& myStringRef = giveMeAString();

  return 0;
}
```

So always check who *owns* the original object the reference if pointing to and when does the original object get destructed.

8.4 Templates with templated template arguments

The GCC's parser for template type name behaves slightly different than the Visual C++ ones. For example this is a valid definition in Visual C++:

```
std::list<std::pair<int, int>> myListOfIntPairs;
```

This is **not** valid while compiling with GCC. You have to separate both > symbols using a space, else, GCC will throw a parser error. So this is the valid equivalent, which compiles on GCC and Visual C++:

```
std::list<std::pair<int, int> > myListOfIntPairs;
```

8.5 Templates using types that depend on the template parameter

When using datatypes in template classes, that depend on the template parameter, GCC needs the additional keyword *typename*. An example:

```
#include <list>

template<class T>
class MyListWrapper {
  // typename is needed here, because the final type of
  // std::list<T>::iterator depends on the template parameter T
  typedef typename std::list<T>::iterator WrappedListIteratorType;
};
```

GCC's error output is not quite clear here (like in so many cases, especially when it comes to templates). Omitting the keyword *typename* in the above code would throw the following compile error:

```
type ‘std::list<T, std::allocator<_CharT> >’ is not derived from type ‘MyListWrapper<T>’
```

8.6 Template types as parameter with default value

I would consider this a GCC parser bug. The following code does not compile on GCC:

```
class MyClass {
  MyClass(std::map<int, int> p_map = std::map<int, int>()) { }
};
```

The error message shows that something seems to go really wrong in the parser:

```
wrong number of template arguments (1, should be 4)
```

The solution is to add brackets around the default argument like this:

```
class MyClass {
  MyClass(std::map<int, int> p_map = (std::map<int, int>())) { }
};
```

Interesting here is, that the error does not occur, if the type has only one obligatory template parameter like std::list for example. So this code compiles fine even without additional brackets on GCC:

```
class MyClass {
  MyClass(std::list<int> p_map = std::list<int>()) { }
};
```

8.7 Member function declarations

When declaring a member function inside the class statement, some people tent to prepend the name of the class to the method name. This may increase readability when inheriting several levels:

```
class A {
public:
  virtual void A::funcFromA();
};

class B : public A {
public:
  virtual void A::funcFromA();
  virtual void B::funcFromB();
};

class C : public B {
public:
  virtual void A::funcFromA();
  virtual void B::funcFromB();
  virtual void C::funcFromC();
};
```

The problem is, this **is not** a valid syntax for GCC. You **must not** prepend the class name to the member function. So the above declaration is valid for GCC like this:

```
class A {
public:
  virtual void funcFromA();
};
```

```
class B : public A {
public:
  virtual void funcFromA();
  virtual void funcFromB();
};

class C : public B {
public:
  virtual void funcFromA();
  virtual void funcFromB();
  virtual void funcFromC();
};
```

8.8 windows.h

You **must not** include windows.h because all the types and functions provided by windows.h are highly Windows specific and will not compile nor run on other platforms. In general you will find the same functionality in QtCore. When using QtCore's functionality, it is easy to compile and run the code on all the platforms supported by Qt.

8.9 for each() vs. foreach() vs. for()

Visual C++ provides a construct which looks like this:

```
for each(int i in myIntList) {
  // loop code here
}
```

This **is not** available on GCC. Therefor this cannot compile on both compilers. But the for each way is handy, so a cross compiler alternative is again the usage of Qt. Qt provides a construct like this:

```
foreach(int i, myIntList) {
  // loop code here
}
```

Using this construct the resulting code is again cross compiler compatible and stays readable and handy. An alternative is always to use the vanilla C++ for(;;) construct.

8.10 stdext vs. __gnu_cxx vs. tr1

Datatypes like the hash_map are currently not part of the C++ Standard Template Library. But compiler vendors provide extensions in their own namespaces. Visual C++ provides this in the stdext namespace, GCC up to version 4.2 in the __gnu_cxx namespace. Since version 4.3 of GCC, the hash_map was moved to the namespace std::tr1 and renamed to unordered_map. The new C++ standard C++0x is on it's way and will contain the unordered_map. So it is very likely that a new namespace will contain unordered_map. For now, we found the following solution to the problem:

```
#if __GNUC__ == 4 && __GNUC_MINOR__ >= 2
#  if __GNUC_MINOR__ == 2
#    include <ext/hash_map>
#    include <ext/hash_set>
#    define HASHMAP_TYPE      __gnu_cxx::hash_map
#    define HASHMULTIMAP_TYPE __gnu_cxx::hash_multimap
#    define HASHSET_TYPE      __gnu_cxx::hash_set
#    define HASHMAP_NAMESPACE_OPEN  namespace __gnu_cxx {
#    define HASHMAP_NAMESPACE_CLOSE }
#  else // __GNUC_MINOR__ > 2
#    include <tr1/unordered_map>
#    include <tr1/unordered_set>
#    define HASHMAP_TYPE      std::tr1::unordered_map
#    define HASHMULTIMAP_TYPE std::tr1::unordered_multimap
#    define HASHSET_TYPE      std::tr1::unordered_set
#    define HASHMAP_NAMESPACE_OPEN  namespace std { namespace tr1 {
#    define HASHMAP_NAMESPACE_CLOSE }}
#  endif
#else // __GNUC__ != 4 && __GNUC_MINOR__ < 2
#  error "unsupported gcc version, need gcc 4.2 or higher"
#endif
```

Yes, this is just the GCC part, to include Visual C++ too, it needs still a bit more code, which is as well included in our branch of CGA. So to keep the cross compiler compatibility the marcos defined above should be used.

8.11 Qt is the key

Qt provides a great way to write platform independent code. QtCore contains lots of things like QThread and QMutex which replace pthreads_create() or WaitForSingleObject() which else would break cross platform compatibility.
So in general, Qt should be used as much as possible to ensure platform independence. Only the unix callmon implentation does not use Qt to keep it light wight. The mechanisms used are highly unix specific anyway.

9 Final words

Keine Ahnung, was genau hier hin soll, aber irgendwie halte ich es für angebracht... :)