AP GCC - CGA tool chain on unix systems

Torsten Becker, Frederik Rudeck, Robert Timm

2009-08-01

Zusammenfassung

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 Alternativ
Implementierung der Callmon Bibliothek unter Linux/MacOS
Wie ist der Callmon-Kern nach Linux/MacOS portiert worden? Wie wurde der Start/Stopp-Mechanismus übertr
Todo
Tutorial: Kontrollfluss-Analyse von Audacity
Charakterisierung des Softwaresystems (LOC, etc.)
Todo
Exemplarisches Vorgehen bei der Faktextraktion (und anschließenden Visualisierung)
Todo

Inhaltsverzeichnis

1 Introduction

2 Challenges

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

2.1 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.

2.2 Platform specifics

The whole CGA tool chain contains 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. Then provided us with a 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 adresses reported by the GCC instrument functions mechanism may be wrong on Linux. Just to name two examples.

2.3 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 function prolog and epilog. The 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. GCC as well does provide this feature, but, it is not supported on x86 platforms. So we had to work around this situation.

2.4 IDE specifics

While introducing the CMake based build system in CGA...

- complex project structure
- custom build commands
- qt custom build commands
- files removed from build
- precompiled headers
- default includes
- const???

3 Results

3.1 CMake build enviroment

3.2 Callmon runtime library

3.3 Patch and Patchclean

3.4 Metacreator

4 Constraints

4.1 GCC Version

need >= 4.2 tested on 4.2, 4.3

4.2 GDB Version

tested on 6.3, 6.8

5 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 and their solution.

5.1 Paths

To be valid on Windows and Unix platforms, a path **must not** contain 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"
```

5.2 Const correctness

5.3 Templates

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 > 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;
```

5.4 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();
};
```

5.5 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.

5.6 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. There 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.

5.7 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 from above should be used.

5.8 Qt is the key

Qt provides a great way to write platform independent code. QtCore contains lots of things which replace pthreads_create() or WaitForSingleObject() which else would break cross platform compatibility.

6 CMake Build System

6.1 Qt specific build steps

6.2 on Windows

6.3 on Linux

6.4 on Mac OS X

7 Unix fact extraction mechanism in detail

7.1 Callmon

7.1.1 Filesystem events

7.1.2 Lockfree event to disk

7.1.3 Asynchronous IO

7.2 Patch and Patchclean

7.2.1 ELF patching

7.2.2 MachO patching

7.3 Metacreator

7.3.1 Line number caching

8 Tutorial

8.1 Preparing the build process

8.1.1 Pitfalls

Compile with -finstrument-functions

Compile with -fno-inline 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 unixcallmonlib as the last library ...

On Linux, link with SYMBOLIC PARAMETER NAME HERE

8.2 Building the application

8.3 Patching the executable

8.3.1 Patchclean

8.3.2 Patch

8.4 Using CGA Toolbar

8.5 Running the application

8.6 Using Metacreator

8.7 Loading the trace(s) into CGA