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 Alternative
Implementierung der Callmon Bibliothek unter Linux/MacOS
Wie ist der Callmon-Kern nach Linux/MacOS portiert worden? Wie wurde der Start/Stopp-Mechanismus übertra
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.

2.2 Platform specifics

The whole CGA tool chain contains lots of platform specific mechanisms like binary patching, collecting of information from the dynamic linker...

2.3 Compiler specifics

The main instrumentation mechanism if based on a feature provided by the compiler. The Microsoft Visual Studio can insert calls to instrumentation function right after a function was called and right before a function returns. The mechanism in general is the same using gcc, but...

2.4 IDE specifics

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

3 Results

3.1 CMake build enviroment

3.1.1 Qt specific build steps

3.2 Callmon runtime library

3.2.1 Filesystem events

3.2.2 Lockfree event to disk

3.2.3 Asynchronous IO

3.3 Patch and Patchclean

3.3.1 ELF patching

3.3.2 MachO patching

3.4 Metacreator

3.4.1 Line number caching

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 Guidelines for GCC and Visual C++ compatibility at the same time

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 other platforms supported by Qt.

5.6 Qt is the key

6 CMake Build System

6.1 on Windows

6.2 on Linux

6.3 on Mac OS X

7 Unix fact extraction mechanism

7.1 Callmon

7.2 Patch and Patchclean

7.3 Metacreator

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