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 **4**

2 Results **5**
2.1 CMake build enviroment . . . 5
2.2 Callmon runtime library . . . 5
2.3 Patch and Patchclean . . . 5
2.4 Metacreator . . . 5

3 Challenges **6**
3.1 Complexity . . . 6
3.2 Platform specifics . . . 6
3.3 Compiler specifics . . . 6
3.4 IDE specifics . . . 6
3.5 Backport to Windows . . . 7
3.6 Mergen . . . 7
3.7 Qt did a great job . . . 7

4 Requirements **8**
4.1 Summary . . . 8
4.2 GCC Version . . . 8
4.3 GDB Version . . . 8
4.4 Binutils . . . 8

5 CMake Build System **9**
5.1 Qt specific build steps . . . 9
5.2 on Windows . . . 9
5.3 on Linux . . . 9
5.4 on Mac OS X . . . 9

6 Unix fact extraction mechanism in detail **10**
6.1 Overview . . . 10
6.2 Callmon . . . 10
6.2.1 Filesystem events . . . 10
6.2.2 Lockfree event to disk . . . 10
6.2.3 Asynchronous IO . . . 10
6.3 Patch and Patchclean . . . 10
6.3.1 ELF patching . . . 10
6.3.2 MachO patching . . . 10
6.4 Metacreator . . . 10
6.4.1 Line number caching . . . 10

7 Guideline - Code that builds on GCC and Visual C++ **11**
7.1 Paths . . . 11
7.2 Const correctness . . . 11
7.3 Returning references to temporaries . . . 11
7.4 Templates with templated template arguments . . . 12
7.5 Templates using types that depend on the template parameter . . . 12
7.6 Template types as parameter with default value . . . 13
7.7 Member function declarations . . . 13
7.8 windows.h . . . 14
7.9 for each() vs. foreach() vs. for() . . . 14

7.10 stdext vs. __gnu_cxx vs. tr1 . . . 14
7.11 Qt is the key . . . 15

8 Tutorial **16**
8.1 Preparing the build process . . . 16
8.1.1 Pitfalls . . . 16
8.2 Building the application . . . 16
8.3 Patching the executable . . . 16
8.3.1 Patchclean . . . 16
8.3.2 Patch . . . 16
8.4 Using CGA Toolbar . . . 16
8.5 Running the application . . . 16
8.6 Using Metacreator . . . 16
8.7 Loading the trace(s) into CGA . . . 16

1 Introduction

2 Results

2.1 CMake build enviroment

2.2 Callmon runtime library

2.3 Patch and Patchclean

2.4 Metacreator

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.

TODO DLLMAIN

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 were missing in the source files which actually needed them. Just to name a few pitfalls.

3.5 Backport to Windows

3.6 Mergen

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

- 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 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.

4.3 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.

4.4 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

5.3 on Linux

5.4 on Mac OS X

6 Unix fact extraction mechanism in detail

6.1 Overview

6.2 Callmon

6.2.1 Filesystem events

6.2.2 Lockfree event to disk

6.2.3 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

7 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.

7.1 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.

7.2 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;
}
```

7.3 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.

7.4 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;
```

7.5 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>’
```

7.6 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>()) { }
};
```

7.7 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();
};
```

7.8 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.

7.9 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.

7.10 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.

7.11 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.

TODO callmon platform specific, rest should use Qt

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