Be More Rational: Open Source Memory Profiling and Performance Tuning

Matthias Kalle Dalheimer

Table of Contents

Open Source Memory Profiling and Performance Tuning

Motivation

Tool support for programmers has come a long way, from the first mnemonic assemblers (and text editors even!) via high-level language compilers, debuggers, cross-referencers to today's complex IDEs and reverse engineering systems.

In general, programmers can choose as many of these tools as they want (or can afford); there are numerous programmers who deliberately or for want of knowledge of more advanced tools use nothing but a text editor and a compiler. But there are two types of problems in software that are close to impossible to fix without advanced tool support:

- Memory corruption problems
- Performance problems

We will look into how open source tools can help you fix these problems in your software.

Fixing Memory Corruption Problems

Memory corruption problems are among those types of software problems that are most difficult to fix, but the right tools can make this ugly and painstaking task simpler and more efficient.

As an example, take the following faulty C++ code snippet:

```
class MyClass
{
public:
   MyClass();
   ~MyClass();

   // ... more methods
```

```
private:
   MyOtherClass* pOtherClass;
   // more instance variables
}


MyClass::MyClass()
{
   pOtherClass = new MyOtherClass();
   // more initialization
}


MyClass::~MyClass()
{
   delete pOtherClass;
   // more destruction
}
```

This looks innocent enough, but as soon as you get assignments or copy construction into the picture, hell will break lose:

```
void myFunc()
{
   MyClass myClass;
   MyClass myClass2 = myClass;
   // more stuff...
}
```

What happens here is that on exit from `myFunc()`, the destructors of both `myClass` and `myClass2` will be invoked, but since both contain a pointer to the same `MyOtherClass` object, this object will be destroyed once (the reason for this being that the compiler will generate a bitwise assignment operator if you fail to provide one; see any good C++ book about this problem which is often called the *Pointer Aliasing Problem*).

Why is this such a bad problem? If you get a crash, you can fire up the debugger and just find the place where it crashed, can't you?

No, you can't. The problem is that these memory corruptions usually do not lead to a crash immediately, but rather a lot (computer-wise) later, when the stack trace will look completely different. Often, the crashes occur when a (typically completely unrelated object) is allocated.

So how do you find those errors? Careful code inspection can achieve that, of course, but we all know how difficult it is to spot errors in your own code. Static automated code inspection tools (that operate on the source code) can do a lot, but these types of defecs can be fairly hidden and therefore would likely not be uncovered.

This is where runtime memory profiling tools come into play. These control your application while it runs and log all memory allocations and accesses. Therefore, it can immediately inform you if a faulty memory operation takes place and log the location in your program where it occurs, even if there is no crash (yet).

Apparently, tools like this are immensely helpful, but also very difficult to write. They have therefore in the past mostly been limited to very expensive closed source tools. Here are some of the tools together with the technology they use:

Rational Purify

Rational Purify uses a patented technique called "Object Code Insertion" which works by changing the executable and library files on-the-fly. The memory audit code is inserted directly into these files, and the "instrumented" libraries are then cached for later use. This means that the executables and libraries need to be reinstrumented after each change to the code, which is time-consuming, but not very cumbersome, since Purify detects this automatically and will perform the necessary instrumentations in a "make-like" fashion.

Purify is fairly thorough at detecting memory problems and also comes with a very good viewer that makes it a lot easier to sift through the generated reports (which can be very lenghty).

Purify's central drawback (besides the high price tag) is platform availability; only the Windows version is really maintained; there are versions for commercial Unix systems that seem to be stuck with very old versions. No Linux version.

Insure++

Insure++ by Parasoft works by replacing the preprocessing stage with a special preprocessor that inserts additional calls to Insure++'s memory audit functions. This has the drawback that upon code changes, the code needs to be both recompiled and relinked with Insure++ which is both time-consuming and cumbersome. Unlike with Purify, you cannot run Insure++ on applications to which you don't have the source code, and to get a full coverage, you need the source code all libraries involved (Insure++ ships with replacements for standard libraries that already contain the instrumentation). This scheme also makes Insure++ fairly fragile to system changes and updates.

Insure++ is very expensive as well, but available on a large number of platforms, including Linux/Intel.

Replacement libraries that replace `malloc()` and `free()`.

There are a number of libraries (both open source and proprietary) that replace the memory allocation/deallocation calls `malloc()` and `free()` with versions of their own that contain additional tracking and housekeeping. By using the `LD_PRELOAD` mechanism, it is ensured that these implementations are used instead of the standard ones.

While these libraries can detect "double deletes" (like in the example above), they cannot detect other types of memory problems, like accessing the tenth element in an array of five elements.

Overloading `operator new` and `operator delete`

C++ allows to overload the two standard operators `operator new` and `operator delete` with versions of your own. There are tools that do just this, and the overloaded versions of course contain additional tracking and housekeeping. The disadvantage of this is that inclusion of the header file with the overloaded version is necessary in each and everywhere source file, so it is an intrusive technique that requires you to change your code first. Also, like with the `malloc()`/`free()` replacement, this technique can only detect a certain type of memory errors.

Valgrind

Valgrind by Julian Seward is a fairly new addition to this list, and has quickly become the memory debugging tool of choice for many open source developers. Valgrind is open source itself and has a comparable functionality to the expensive proprietary tools. Valgrind works by implementing a virtual machine in which the application is run. This virtual machine tracks all memory accesses and can thus notify you about memory corruption problems on the spot.

The big advantage of this technique is that you can run applications under Valgrind's control at any time, without any preparation. We will have more to say about Valgrind below.

Valgrind

As already mentioned, Valgrind is an excellent tool for debugging memory corruption problems. You simply prepend `valgrind` to the application command line, and your application will run as usual (albeit a lot slower).

Here is an example of Valgrind output:

```
==22173== Memcheck, a.k.a. Valgrind, a memory error detector for x86-linux.
==22173== Copyright (C) 2002, and GNU GPL'd, by Julian Seward.
==22173== Using valgrind-1.9.3, a program instrumentation system for x86-linux.
==22173== Copyright (C) 2000-2002, and GNU GPL'd, by Julian Seward.
==22173== Estimated CPU clock rate is 502 MHz
==22173== For more details, rerun with: -v
==22173==
==22173== Invalid free() / delete / delete[]
==22173==   at 0x4016898F: free
(/home/kalle/valgrind-1.9.3/coregrind/vg_clientfuncs.c:182)
==22173==   by 0x80484BB: main
(/home/kalle/valgrind-1.9.3/memcheck/tests/doublefree.c:10)
==22173==   by 0x402439EC: __libc_start_main (in /lib/libc.so.6)
==22173==   by 0x80483B0: (within
/home/kalle/valgrind-1.9.3/memcheck/tests/doublefree)
==22173==   Address 0x40F53024 is 0 bytes inside a block of size 177 free'd
==22173==   at 0x4016898F: free
(/home/kalle/valgrind-1.9.3/coregrind/vg_clientfuncs.c:182)
==22173==   by 0x80484BB: main
(/home/kalle/valgrind-1.9.3/memcheck/tests/doublefree.c:10)
==22173==   by 0x402439EC: __libc_start_main (in /lib/libc.so.6)
==22173==   by 0x80483B0: (within
/home/kalle/valgrind-1.9.3/memcheck/tests/doublefree)
==22173==
==22173== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==22173== malloc/free: in use at exit: 0 bytes in 0 blocks.
==22173== malloc/free: 1 allocs, 2 frees, 177 bytes allocated.
==22173== For a detailed leak analysis,  rerun with: --leak-check=yes
==22173== For counts of detected errors, rerun with: -v
```

This comes from a "double delete" error, only that `free()` was used here instead of `operator delete`. Together with the stack trace information including the filenames and line numbers, this makes it fairly simple to find the trouble spot.

However, Valgrind output rarely is as simple and obvious as in this case. Often, it is many hundreds or thousands of lines long, including a number of "false positives". These often stem from third-party libraries like X11 that you cannot or do not want to debug. For this, there are so-called suppressions, however, configuring those suppressions is very difficult at this time. This is one of the areas where the proprietary tools still have an advantage. However, open source developers (including the author) are already working on remedies to this problem.

Profiling

The other software development task that is difficult to do without software tools is optimizing an application. Actually, it is usually not so difficult to take any arbitrary piece of code from your application and make it run faster; the problem is how to find the spots that actually need optimization - the so-called inner loops - so that you do not spend time optimizing code that is rarely executed or simply not a performance bottleneck at all. This task is called *profiling*. In profiling, you want to find out which functions or methods are called most often and which have the longest runtime (and particularly of course which functions or methods are called often *and* have a long runtime).

In Linux development, **gprof** is the traditional profiling command. However, gprof's output is neither very comprehensive nor very easy to understand. Probably the best profiler on the market is Rational Quantify which

employs the same code-instrumenting technique as Rational Purify mentioned above, and which also has a very good viewer, but suffers from the same problems (price tag and platform availability) as Purify.

Again, Valgrind comes to the rescue. Because of the virtual machine technique, Valgrind can actually be used for more than just finding memory problems. An additional module, a so-called *skin*, called *cachegrind* tracks the number and duration of all function and method calls and thus gives you valuable data about where the performance bottleneck in your code are.

While Valgrind provides more data than gprof, it would only be a small improvement if it was not for KCacheGrind, a KDE frontend to cachegrind. This provides very simple navigation to the collected data and makes it really easy to locate the performance bottlenecks.

Summary

With the advent of Valgrind and its various skins like cachegrind, high-quality tools for memory debugging and profiling are now available to the open-source community. Frontends like KCacheGrind make these tools more accessible for software developers.