IgProf profiling tool

G. Eulisse, L. A. Tuura
Northeastern University, Boston, USA

Abstract

A fundamental part of software development is to detect and analyse weak spots of the programs to guide optimisation efforts. *valgrind* and *oprofile* are two excellent tools that provide developers detailed information about their programs. However they still leave holes to be filled. *IgProf* is a new flexible debugging and profiling tool that can analyse large and complex applications.

IgProf has three main components: a core profiler built on top of a generic event collector and dynamic function instrumentation tool, a number of profiling modules, and a utility to analyse the gathered data. This article describes the profiler's features, output and implementation.

INTRODUCTION

Three typical programming problems, immediately after being unable to compile or run the program in the first place, are: *it's too slow*, and *it uses too much memory*, and *it's both too slow* **and** *uses more memory than any realistic machine has.* The obvious solutions of buying a faster computer, more memory, or a faster computer with more memory often yield only limited and passing relief: the capacity of programmers to consume resources frequently exceeds not only the supply but also the growth of the supply of those resources. The present shortage of well-established enemy nations and lavish government grants has caused additional funding shortfalls.

The developer is therefore faced with a major ordeal: debugging and optimising the code.

Profiling

Optimization and debugging are tedious work to be done; some even report it to be impossible.[1] To ease this painful task a number of tools have been developed and new ones are being continuously written. To name just a few, *gdb*, *valgrind* and *oprofile*, cover a large spectrum of what is generally needed to debug and profile our code.

It is crucial to have excellent profiling tools in order for a developer to be able to focus on real, not imagined problems. It is not worth optimising a piece of code which only takes 1% of the execution time or memory if some other part consumes 35% of them. It is important that the tools point developers quickly and effectively to the real source of the problems.

Profiling a program gives a quantified answer to the question "How much does each function in this program consume resource X?" The resource may be for example run-time (performance profile), amount of memory allocated (memory profile), thread locking primitives (lock contention profile), or system resources such as file descriptors. If the program should free the resources is has acquired, such as memory, profiling can also tell where resources are leaked.

A common approach to performance profiling is event sampling: the program execution location is sampled every once in a while, either through instrumenting the code (e.g. at compile time), or by statistical sampling (e.g. `SIGPROF` signal or non-maskable hardware interrupt). The position within the program is recorded and stored in a data structure which may be either hierarchical or flat. Over a long run period the distribution of these samples produces a fairly reliable indication where most of the time was spent.

Resource profiling is in fact analogous to performance profiling. The events to be sampled are calls from the program to acquire and release resources, for instance to `malloc()` and `free()` memory management functions. The only difference is the profiler counts memory allocated and freed, not just the number of "ticks."

Existing Open-Source Tools

Before describing IgProf we shall briefly cover a number of existing open-source tools. They have been an inspiration in developing our own tool; we have sought to complement, not duplicate their functionality. In fact IgProf started as an attempt to improve some of the existing tools.

valgrind Valgrind [7] is a memory debugger for Linux systems. It only supports IA32 (i386-family) architecture. At program start-up it captures the execution and runs the program on a virtual IA32 processor. It can be used with almost any ELF-based executable. By simulating the whole CPU, valgrind is able to track every memory operation and track every bit written to and read. This information is used to generate warnings when uninitialised values are used, accesses outside valid memory and leaks. These days the tool has been extended to support an impressive range of "skins" from cache performance analysis to discovering synchronisation race conditions.

[1] Bruce Leverett is reported to have written in "Register Allocation in Optimizing Compilers": But in our enthusiasm, we could not resist a radical overhaul of the system, in which all of its major weaknesses have been exposed, analyzed, and replaced with new weaknesses.

oprofile Oprofile [6] is a kernel level profiling facility. It uses debugging event registers that exist on most modern CPUs; when a predetermined count of events is exceeded, the CPU generates a non-maskable interrupt. Oprofile kernel module records in which program and where in that program the event occurred. By sampling the system and the programs for a while a fairly accurate profile is generated. The choice of events that can be monitored is very rich on modern CPUs and includes a variety of hardware-level details.

jprof jprof [10] is a small performance profiler developed by the Mozilla [9] project to allow the browser to be profiled by itself. It uses standard `SIGPROF` signals for sampling and can be injected into programs with the `LD_PRELOAD` mechanism, thus requiring no instrumentation.

Requirements for a profiling tool

As a memory debugger, valgrind does an excellent job in finding memory leaks and general memory errors. It can however be too slow for computationally intensive applications and while there are plans to provide quantitave information on memory usage and allocations, these are unavailable at this time. Moreover, it is only available for x86 systems and limited to the instruction set present in the emulation. Oprofile currently lacks a hierarchical profile data view and requires administrative privileges to enable the profiler. This in practise prevents profiling on computing farms as farm administrators rarely allow non-standard kernel extensions to be loaded. Jprof as completely unintrusive profiler is better suited for farms, but lacks many features we would like to have; IgProf grew out from extending and rewriting jprof. Here it is our list for the “most wanted” features of our “ultimate profiling utility”

- “Fast enough to profile large software systems used by LHC experiments”
- “Capable of dealing with threads, shared libraries and dynamic loading”
- “Unintrusive: no compile- or link-time instrumentation”
- “Very easy to use, robust”
- “Profiler itself must not skew profile data”
- “Able to profile performance, both real-time and process-time”
- “Able to profile memory usage”
- “Fast resource leak detection”
- “Must generate both flat and call-tree profiles”
- “Extensible to allow new types of events to be sampled”

Most of the low level features needed to gather such debugging/profiling information are already present in the system libraries such as *glibc*. What is left for the profiler itself, at least to the first order of approximation, is gathering profiling events, tracking them in an optimised structure at run-time and producing nice output that is easy enough to interpret and navigate. This is how how *IgProf*, the ignominous profiler, was born.

General structure of the profiler

The *IgProf* profiler is injected into the target program normally as an auxiliary library at program start-up time. The core profiler provides services to observe and track profiling “events” plus all the infrastructure to track thread creation and exit and to output the profile data at program exit. The profiling events include a stack trace at the point the event occurred plus some event-specific data. An event can be a point in time or mark a time-span. In the latter case the event is made of two parts: a start and end notification.

Once the profiler is attached to the program, the selected profiler modules are activated. The modules define which events are gathered and usually depend on the options defined by the user: memory allocations and deallocations, timer interrupts, instrumented calls and so on. Each module activates, handles and filters its own events, using the core services. Typical event captures the current call stack and inserts an event into a compact tree that accumulates the results.

We use the compact accumulating tree to avoid tracking every single event in full detail. It is sufficient for profiling instantaneous events such as bytes allocated by a function or number of hits in performance sampling. Additional “live” resource tables are used when tracking leaks or maximum live resource usage. These hashed resource tables track events that have started but not yet ended, for instance after `malloc()` was called to allocate memory and before `free()` was called to release it. When the end event occurs, the accumulated tree is adjusted accordingly. When the program ends, entries left in the resource tables are leaks. Other profiling tools can be used for more precise performance sampling when aggregated results are not enough.

IGPROF WORKFLOW

The IgProf user interface is very similar to valgrind: user runs the program with `igprof` prepended to the command line. The command is actually a wrapper script that forces the profiler to be injected to the target program and to activate before the target program gets to execute. The profiler does its instrumentation on the target program, for instance for memory profiling instruments the `malloc()` and `free()` functions, and then lets the program execute normally. When the instrumented functions are called or for instance SIGPROF signal is generated, a profiling event is generated and the profiling core gets informed about it. The information about the position at which the event occurred is saved in a tree like structure. When the program exits the profiling information are dumped to the disc in the form of an XML file. At this point an external utility can be used to transform this rawtrace XML file to a more significant layout. In the IgProf package we provide a simple utility "igprof-analyse" that allows to convert the XML file into a gprof like output file.

Hooking mechanism

IgProf uses dynamic hooking to instrument the target program. That is, instead of overriding dynamic linker symbols for the functions to be instrumented, IgProf looks up the original functions at run-time and replaces the function prologue with a jump to a profiling trampoline. The trampoline has two parts: a jump to the instrumentation hook, and copy of the original function prologue. Hence any call to the original function will end up directly in the hook function. The hook function typically then calls the original function by calling the second part of the trampoline, which executes the copied original prologue and then jumps into the next instruction in the original function. This technique was originally described in [1] and similar technique is used in a number of projects [2, 3, 4, 5].

This mechanism allows IgProf to decide at run-time which functions should be instrumented, and to do so relatively independent of dynamic linker and system library versions. There is no danger of hook function clashes and inadvertent recursion. The hook function has complete access to the parameters and return value of the instrumented function. Obviously there are uses of the hooking mechanism beyond purely profiling, and hence we provide the code in a separate IgHook package.

Possible Measurements

IgProf can currently profile performance (`-pp` option), memory usage (`-mp`) and file descriptor usage (`-fd`). The memory profiler can track the total amount of memory allocated, the largest block of memory allocated, the amount of live memory left at exit, and the maximum live memory at any point. In leak check mode (`-cl`) all live blocks at the end are reported, with the live memory at the end counts indicating how much each function leaked memory. The file descriptor profiler can track the number of file descriptors used, live descriptors at exit (leaks), and maximum number of live descriptors at any point.

Any number of these profiles can be enabled simultaneously—it is perfectly viable to enable all of them at once. This is particularly useful if the program is very large and takes a long time to run, for instance several hours or days. Enabling more than one profiler module of course introduces some skew as each profiler consume resources themselves. The core profiler disables all accounting while within any profiler modules, so the effect is minimal, but it still happens that the performance profiler generates hits on other modules.

Interpreting the Results

Once filtered with `igrof-analyse` IgProf output resembles that of gprof so its output conventions documentation may be helpful to read first [8].

The following explanation assumes memory leak checking. In the case of measuring something else, the explanation has to be adjusted correspondingly for CPU time or memory allocated. The output format is the same, just the meaning of the numbers is different.

The result file is made of sections like shown in Figure .

Each section describes the statistics for one function, the primary function, surrounded by secondary functions. Above the primary function are the callers: the functions that called the primary function. Below the primary function are the callees: the functions the primary function called. The line that begins with a bracketed number, here '[0]', indicates the primary function; the number in the brackets indicates the function's index in the statistics. The smaller the index is, the higher the function was in cpu time usage, memory usage or leakage. The next columns are the statistics explained in more detail below. The first text column tells the name of the function, here JetFinderEcalPlusHcalTowerInput::prepareInput(RecEvent const*), followed by the name of the module where it was found, here writeDST. The callers and callees give the index of the function after the module name, such as the '[952]' and '[732]' in the first two callees here, to help in navigating in the output.

For the primary function line the first statistic is the sum of the memory leaked by the function and all its callees, here 50642044. The second statistic is the amount of memory leaked directly by the function itself, here 50226044. So in this case we can already see that most of the memory was leaked by the function itself. The third statistic is the number of unique call paths that resulted in a leak, here 5. Note that this is not the number of calls that resulted in a leak, but the number of unique call paths: there were five different calls stacks with this function in the stack. There could have been thousands of such calls.

For the secondary functions the statistics are slightly different. The first statistic is the total amount of memory the function leaked in any call path. The second statistic is the amount leaked when the primary function was also in the call stack. For callers this is the amount of memory the caller leaked through the primary function. For callees it is the amount the primary function leaked through that callee. So the second number is always less than the first one. The third number pair is first the number of calls paths including the primary function, and then total number of calls to the function.

RESULTS

IgProf has been used in a number of cases to spot memory and performance issues of CMS recontruction software. It has been expecially important in the phase preceeding CMS DC04 to tune the DST writing software and

```
     53553180   50642044   5/5     PersistentJetFinder::reconstruct() (.../ORCA_8_0_1_pre1/lib/...)
 [0] 50642044   50226044   5     JetFinderEcalPlusHcalTowerInput::prepareInput(RecEvent const*) (...)
      5863168          0   5/62    BaseRecItr::BaseRecItr[not-in-charge](RecoQuery const&) (writeDST) [952]
    229962815          0   5/49    LazyObserver<RecEvent const*, int>::check() const (writeDST) [732]
        36864          0   5/14    std::vector<VJetableObject*, std::... >::_M_insert_aux(...
            0          0   5/5     JetableObject<EcalPlusHcalTower>::JetableObject[in-charge](...) (...)
     62299760     416000   5/547   RecQuery::operator RecConfig const&() const (.../COBRA_7_7_1_...)
```

Figure 1: Sample igprof-analyse output

remove two big memory leaks that would have seriously compromised the ability to run the DST production as forecasted. In particular a 180Mb per 1000 events leak in the muon code and a 20Mb/1000 events one in the tracker code have been exposed and fixed thanks to it.

CONCLUSION

IgProf has been proven to be a useful tool to improve the quality of CMS simulation and reconstruction software. The profiling core is stable, reliable and fast enough to profile programs requiring even 500Mb of RAM. A number of weak spots have been found easily and fixed thanks to it, expecially for what concerns memory leaks. We look forward in trying to continue improving the user frontend which is currently the real weak spot of the profiling process.

ACKNOWLEDGEMENTS

This work is supported by NSF.

REFERENCES

[1] Jeffrey Richter, “Load Your 32-bit DLL into Another Process’s Address Space Using INJLIB”, Windows System Journal, Vol 9 No 5, May 1994.

[2] Shaun Clowes, “injectso: Modifying and Spying on Running Processes Under Linux and Solaris”, The Black Hat Briefings, 2001, Amsterdam, http://www.blackhat.com/html/bh-europe-01/bh-europe-01-speakers.html#Shaun, http://www.securereality.com.au/

[3] “DynInst: An Application Program Interface (API) for Runtime Code Generation”, http://www.dyninst.org/

[4] http://rentzsch.com/mach_inject/

[5] http://rentzsch.com/mach_override/

[6] http://oprofile.sourceforge.net

[7] http://valgrind.kde.org

[8] http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_node/gprof_10.html

[9] http://www.mozilla.org

[10] http://www.mozilla.org/performance/jprof.html