Performance Analysis of GUI applications on Linux

Josef Weidendorfer

Submitted as Whitepaper for a talk at the KDE developer conference "Kastle 2003"

Abstract

Optimizing performance of a regular Linux application is a difficult task using the legacy profiling tools like GProf, as one needs special prepared static linked binaries. This is especially a problem with modern GUI applications typically using a large number of shared libraries and dynamically opened plugins.

This paper presents two tools, Calltree and KCachegrind, which help the developer in doing performance analysis: the first is a profiling tool with a cache simulator and call graph tracer, able to run with unmodified binaries by using runtime instrumentation. The second is a visualisation tool for comfortable browsing the huge amount of data produced in a profile run.

Unlike numerical applications, the event driven nature of GUI applications poses problems for profile visualisation: apparent mutual recursion of functions probibit meaningful interpreting of profiling data. Some solutions for this problem are presented.

1 Introduction

With the performance of modern CPUs in mind, it seems a waste of time to put much development effort into optimization. But this attitude, combined with increasing use of abstraction layers to cope with complexity, can quickly lead to slowdowns in software the hardware can not compensate for. This is especially true for GUI applications: drawing functions happen typically at a stack depth of 30 and more in a KDE application, and that's only the client side handling without X server. It is almost unpossible to estimate the effort needed to accomplish an update request. Thus, profiling is needed to get knowledge on the performance characteristics of an application.

But profiling a Linux application is a difficult task using the legacy profiling tools like GProf. The use of shared libraries and dynamically opened plug-ins even prohibits its use at all. The result is that in most cases, performance optimization doesn't happen. If there's an obvious performance problem, one has to estimate the location of the bottleneck or resort to handcrafted meassurement routines.

Last year, a tool for instrumenting binary code on the fly was released for Linux under the GPL, called Valgrind [3]. Using the technique first as a memory debugging, it was quickly adopted by the open-source community for its ease of use. Valgrinds usefullness for program execution analysis is less known, but likewise powerful. There is the cache simulation skin[1] "cachegrind", delivering flat profiling on a source line level and accumulated for functions. Still, to get a real overview of the performance characteristics of a program, one wants to see the meassured costs attributed to callers of functions to get *inclusive* costs, i.e. cost of a function including the cost of all the functions called from there. This is especially important for programs doing the real work in 3rd-party libraries with long call chains like GUI applications. Thus, call tracing was added to "cachegrind", resulting in the tool "calltree".

To make use of the huge amount of data delivered by this tool, a visualisation allowing for quick browsing at different granularities ranging from costs of instructions to summed up costs of a whole shared library is needed. A program delivering these features is KCachegrind, likewise released under the GPL.

In chapter 2, an overview of techniques used for profiling in general is given. Chapter 3 con-

[1] The term "skin" denotes a Valgrind core extension for specific supervision actions. When starting Valgrind, you specify the skin to use. Default extension is "memcheck", the memory debugger. This technique is kind of a plugin architecture: skins are shared libraries, possible delivered in a separate distribution from Valgrind itself.

tinues with an description of tools available for Linux, taking a more detailed look at the mechanisms used in Valgrind, Cachegrind and Callgrind. Chapter 4 shows the requirements needed for a visualisation tool to enable successfull performance analysis, together with the featureset of KCachegrind. The paper finishes by sketching use cases of the presented toolchain.

2 Profiling an Application

The first step in optimizing code is to look for the code ranges where most of the time is spent: by optimizing these program areas, one will get the best performance benefit for a given developing time invested into optimization. To this end, the program is run under the supervision of a tool which is collecting performance properties.

2.1 Objectives of Profiling

An exact *trace* of executions of functions and calls among them is given in a dynamic call-tree (DCT), see fig. 1 (a). Calls from a given function are done in left to right order, and the size of the tree depends on the number of calls done. Gathering that much information is unneeded for the given purpose. A *profile* of a program run results in a dynamic call graph (DCG), where each function only appears as one node, and every call to/from this function is depicted by an arrow to/from this node. Fig. 1 (b) shows the corresponding call graph to the previous given call tree[2]. Dynamic call trees and graphs are generated by calls happening in one specific invocation of a program. In contrast, a static tree or graph is generated by static analysis of the program code and ideally represents all calls possible in a program[3]. We only talk about dynamic graphs, so we leave out "dynamic" in the rest of this paper.

In converting a call tree into a call graph, information is lost. E.g., the call graph in fig. 1 (b) doesn't represent the fact that function C is not called from B when D is active. A compromise is a context call tree (CCT) [4], given in

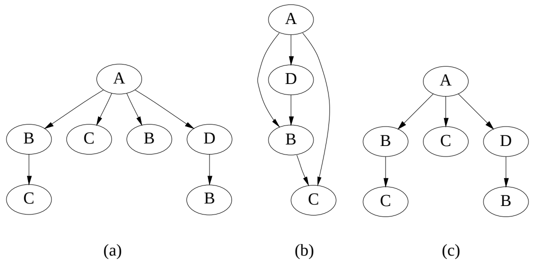

Figure 1: A dynamic call tree, its call graph, and its context call tree

fig. 1 (c). Here, only identical call chains from the root function are collapsed, yet resulting in a size limited by code size instead of run time[4].

It should be noted here that for performance analysis, a more detailed look at the control flow of a program than the pure call graph can lead to better insight. Thus, it can be worthful to distinguish different control flows inside of a function. Motivation for this is the insight that with modern, out of order, superscalar CPUs, events are happening dependent on the current instruction stream, and often can not be related to the execution of a single instruction. Legacy profiling tools like GProf still try to provide such details, and only confuse the user this way[5]. Still, techniques for instruction stream profiling are subject to research. So we will concentrate on inter-function relationship only.

Attributed Call Graphs

Profiling an application aims in getting an attributed call graph of a given run. Attributes of interest for functions (nodes of the graph) are

- the number of events happening while running inside the function (*exclusive* or *self* cost),
- the number of events happening while running inside the function and all the functions which are called from the given function (*inclusive* or *cumulative* cost),

and for calls (edges of the graph)

[2]The presented profiling tool "calltree" produces call graphs and its name is therefore misleading. An rename to "callgrind" is pending to better show its relation to Valgrind.

[3]A comparation of a dynamic with a static call graph allows for checks for complete code coverage of application regression tests.

[4]It is assumed here that mutual recursive functions are treated special to not produce contexts in the number of the maximal recursion level reached in a run.

[5]In contradiction to this, Calltree attributes costs to instructions. As only simulation of the cache hierarchy is done and not of branch prediction or out of order execution, this still makes sense. Besides, trustworthiness of the tool is increased by such details, as the synthetic events are deterministic and understandable.

- the number of events happening while the program is running in code called via this call arc,
- the number of times the call happens.

Assume the call tree of fig. 1 (a) and an abstract event type. In every function, one such event happens, i.e. every function invocation has an exclusive cost of 1. The attributed call tree with events spent in each call is given in fig. 2 (a). The corresponding call graph is shown in fig. 2 (b). The call counts are shown above event sum on the calls, which were given implicit in the call tree (always 1).

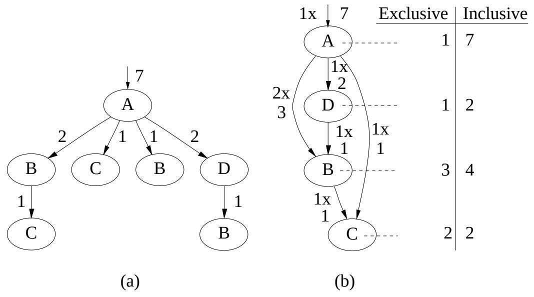

Figure 2: Attributed call tree and its attributed call graph

Events can be memory accesses, cache misses or just timer ticks passed by while executing a range of the code. Event types other than timer ticks can give some hints on why time has been spent. E.g. by looking for 2nd-level cache miss events, the time spent in fetching data from main memory can be estimated. If compiler generated code isn't smart enough to hide this latency (e.g. by doing cache prefetching), performance most probably will get better by code modifications leading to reduced numbers of cache misses. Another example of an interesting event type would be the number of floating point instructions executed in a time unit. By this, one can check if the program exploits the floating point peak performance of the CPU.

One should choose representative input data for a program when doing a profile run to get typical performance characteristics. Also, the program code and the runtime environment should be the same as the environment where the application will be run by the end user. Aside, one has to be cautious if profiling results are only limited to the user level space of a program run. This can be misleading when a lot of I/O is done, or when the program depends on external applications like in client-server architectures. In this case, system-wide profiling should be done.

How does profiling actually help in improving a programs code for better performance?

- A programmer has assumptions on the runtime behaviour of his code, like how often a given function will be called. By profiling, one can check if these assumptions hold true.
- To get the same output of a program, there often exist different algorithms and data structure layouts which can be used. By comparing the outcome of a profile run against these different alternatives, one can choose the version leading to best performance.
- If a general performance improvement is to be reached for a given program part, profiling can output the source code ranges where most of the time is spent, and thus, maks suggestions for most effective optimization effort.

2.2 Profiling Methods

To get the needed information, a profiling tool has to do some measurements. These measurements should not perturb the performance characteristics of the original program to be useful at all, i.e. profiling measurement has to influence the behaviour of a given program in a minimal way.

The best way to gather the data is to use special hardware. Modern CPU architectures include performance counters for a large range of event types to enable profiling. Reacting on every event happening would need extensive hardware resources. A compromise is statistical sampling.

2.2.1 Statistical sampling

For the results of the statistical sampling approach to be true, the following assumption should apply as good as possible: the distribution of every n-th event happening over the code range of a program is the same as the distribution of every event of same type. Alias effects aside, and according to the law of big numbers, this assumption would be met by running a program an infinitive number of times in the row on the same input. In practice, a single run is often enough.

The hardware needed for this approach is simple: there exists a counter register for a given event type, which can be set to an initial value. This counter is decremented when the event happens. When the counter reaches zero, an interrupt is raised. The interrupt handler has access to the program counter of the interrupted instruction, updates counters related to this address, resets the hardware performance counter and resumes the original program. If the inital value is choosen to be large enough, the time spent in the interrupt handler should be neglectable. Note that the larger the initial value is, the less accurate the profiling result will be.

The result of statistical sampling is self time of code ranges. If no further measurement is done, heuristics have to be applied to get the attributed call graph mentioned above: A static analysis of the program code gives the call relation among functions. Note that targets of jump/call tables can't be detected this way (used e.g. for C++ virtual methods or switch/case statements on enumeration types). The number of calls happening can't be estimated at all; estimations of inclusive time is only vague because the cost distribution to calls to a given function has to be estimated, too. Still, for FORTRAN programs, profile information gathered this way is often enough to improve performance.

2.2.2 Instrumentation

To overcome the limits of pure statistical sampling, one has to instrument the program code to be profiled. Several instrumentation methods exist: Source modification, compiler injected instrumentation, binary rewriting to get an instrumented version of an executable, and binary translation at runtime. The last method is the most comfortable for the developer, as unmodified executables and shared libraries can be used.

Instrumentation adds code to increment counters at function entry/exit, reading hardware performance counters, or even simulate hardware to get synthetic event counts. To minimize measurement overhead, often only small action is performed. Even then, e.g. GProf instrumentation still can have an overhead of 100%. An interesting approach is Ephemeral Instrumentation [11]. It tries to combine the advantage of statistical sampling - whose overhead can be adjusted according needs - with instrumentation by running the instrumented version of functions only in short, periodical time intervals. This is done by instrumenting conditional branches periodically on the fly at runtime; the instrumentation removes itself after a number of executions.

If synthetic event counters, generated by a hardware simulator, are enough for the profiling result quality to be achieved, runtime instrumentation with its ease of use is possible. The instrumentation engine and the instrumentation itself don't have any influence on event counters: these aren't meassured but "calculated". Complex hardware simulation can be done online and only practical issues limit the possibilities: No one will use a tool with a slowdown factor in the thousands. An issue one always have to keep in mind here is, how good the produced synthetic results will match reality.

2.3 Visualisation of Mutual Recursion

One issue appearing when profiling large applications, especially object oriented and/or event driven code, is the problem of true or pretended (false) mutual recursion of functions in the generated call graph, so called *cycles*. Fig. 3 (a) gives a modified call tree of Fig. 1, leading to a true and a false cycle in the call graph: there seems to be the mutual recursion $\langle D > B > C > D \rangle$ now.

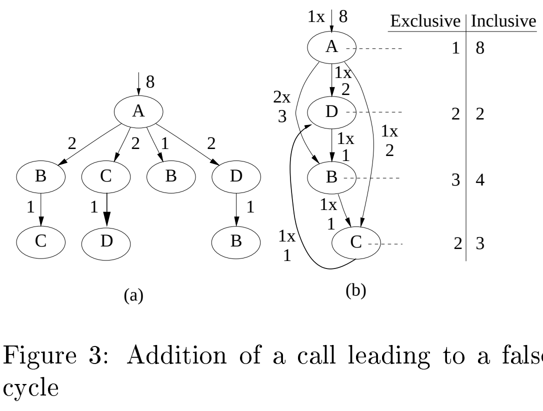

Figure 3: Addition of a call leading to a false cycle

Typically, profile tools only provide a call graph. Thus, a visualisation tool can't distinguish the two types of mutual recursion. Moreover, in the case of e.g. GProf, the tool only generates call counts. The heuristic used for calculation of call costs depends on a topological sort for cost aggregation from the bottom to the top. Inside of true or false cycles in the graph, costs distribution can't be calculated.

On the other hand, even if the tool has collected call costs, these are of limited use to the developer. As an edge in a cycle of a graph can possibly appear multiple times in a given call chain (like the edge B/C in $\langle A > B > C > D > B > C\rangle$ of the given example, assuming an attribution with multiple calls along B/C), the summed cost of an edge can be potentially greater than the run time of the whole program. Usually, inclusive cost of a function is the sum of self cost plus the cost of outgoing calls. In the case of cycle members, this can't hold true, as we can get inclusive cost larger than program run time this way!

For the set of functions a cycle consists of, we can introduce a new temporary function for the purpose of presentation, with the following rules:

- The self cost of the cycle is the sum of self costs of its members.
- The inclusive cost of cycle members is the sum of self cost of this member plus the cost of outgoing calls leaving the cycle.
- The inclusive cost of the cycle is the sum of inclusive costs of its members.

This gives reasonable values and is analogous to the fact that calls representing direct recursion likewise has to be ignored for calculation of the inclusive cost of a function. It has to be noted that this cycle visualisation is only of limited value, as the interpretation of inclusive costs of cycle members is somewhat screwed: The original meaning of "costs of this function and all function called from here" no longer holds true, as inner-cycle calls are skipped. Within cycles, there exists no cost relationships among functions.

Unfortunately, in real world bject oriented code or GUI applications, one always will find large cycles because of polymorphic language features or event driven programming: most call chains will go over a few centric dispatching functions in the code, thus producing large, false cycles. In this case, an visualisation issue is to first give an quick overview of the structure of big cycles: one wants to understand why a cycle was produced at all. Collapsing "uninteresting" cycle parts to get a rough, but small view will be needed. This is in need of some research.

2.4 Cycle Avoidance

The logical solution for the cycle problem presented in the last chapter is a change in the profiling tool to not generate a call graph with false cycles in the first place. Second, true cycles eventually should be "unrolled" to a given degree.

Call Chain Contexts

The solution for avoidance of false cycles is given at the start of this chapter. If the tool doesn't generate a call graph, but a context call tree, false cycles won't appear. The theoretical number of possible existing call chains in a CCT seems to be way too large for practical purposes. Instead, we go back to a call graph, but distinguish functions according to a call chain with limited length. This way, we define multiple contexts for each function. This can reduce the number and/or size of false cycles.

But it also increases the amount of data produced. Table 1 shows the number of contexts for different applications with various limits on the length of the call chain[6]. For all applications, the number of contexts and calls among contexts converge, but for large applications, the needed call chain length is out of reach regarding needed memory resources of the profiling tool used here. E.g. for konqueror, with limit 10 the process has a maximum size of 553 MB, for limit 10 of 658 MB. The ratio among number of context with limit 1 and limit 0 is a hint for code reusage in the application; the compiler code seem to do best in this regard.

The results prove that it seems not possible to get all contexts for a modern GUI application with a tool like calltree on a 32bit machine because of memory requirements.

A similar approach is to only select a few functions of a call chain for distinuishing contexts. With a good heuristic, this should prohibit false

[6] On Suse 8.0 with KDE 3.1.0 and QT 3.1.1. Athlon MP 1.54 GHz with 1GB memory. Context numbers produced by Calltree 0.9.3 (with –simulate-cache=no to lower memory needs). Mutual recursion is distinguished until level 2. ct_main-i.c is the main source of Calltree after preprocessing. For the last 2 programs QT Style Windows was choosen, for Konqueror the pure Iconview. Function count is an average for GUI applications.

Command		Call chain length limit 0	1	2	5	10	20
true	Contexts	296	612	700	887	891	891
	Calls	404	489	588	727	728	728
bzip2 libm.so.6	Contexts	408	850	1 004	1 329	1 332	1 132
		538	688	861	1 113	1 113	1 113
xclock	Contexts	2 374	5 311	7 004	10 961	11 588	11 741
		3 826	5 525	7 446	10 228	10 699	10 783
cc1 ct_main-i.c	Contexts	1 519	5 157	8 352	22 060	41 164	44 899
		6 741	10 881	15 905	34 282	52 191	54 722
qtconfig	Contexts	9 980	26 657	43 039	116 677	194 221	228 276
		24 985	42 818	68 788	147 051	216 843	245 606
konqueror /lib	Contexts	21 500	55 829	91 713	251 449	420 871	507 507
		51 052	90 629	147 958	315 838	470 032	544 487

Table 1: Number of contexts and distinct calls with various limits on the caller chain length.

cycles, but minimize the data produced in the profile run. The drawback is that for best function selection, one has to use feedback information from an previous, similar run. This could be a practical problem.

Ignoring Function Invocations

We already stated that for GUI applications, there exist a few centric functions which seem to be responsable for the existance of most false cycles in a call graph. By ignoring these functions, we can't get cost measurements for them, but effectively avoid a lot of cycles, improving the call graph quality.

Distinguishing Recursion Levels

For cycles caused by real mutual recursion, we can distinguish functions according recursion levels. For example, let us assume a call chain A > B > A > B > A. Using the recursion level as context, we effectively get new function names in the call chain A'1 > B'1 > A'2 > B'2 > A'3.

Theoretically, this can be done transparently for the visualisation tool. In practice, it makes sense to also present summed costs for the original functions A and B.

Lossy Cycle Detection

For correct behaviour regarding cycles, the visualisation tool has to detect strong connected components in the call graph. By introducing a heuristic into this detection, the number of cycle members can be reduced: We ignore all edges with cost attributes below a given threadshould in the cycle detection algorithm. Experience with GUI code shows that often, a call chain happening only 1 time for initialisation purposes enlarges a false cycle by a great amount.

Sometimes, this heuristic can go wrong and lead to functions with cumulative costs larger than 100% of the total cost, especially for functions involved in real recursion.

As a summary for this section, it has to be noted that the ideas presented are subject to work in progress.

3 Profiling Tools

3.1 Overview of Tools Available for Linux

The most known profiling tool on Linux is GProf [5]. The corresponding command is the postprocessing tool for "gmon.out" files, generated by binaries compiled with GCC, giving the option "-pg". This option instruments function entries to calculate counts for call arcs executed, and enables statistical sampling using a timer signal raised every 1/100 second. Measurements inside of shared library code is not supported. Therefore, profiling only makes sense for static binaries or when you are sure that by far most of the time is spent in functions of your own executable. But even then, the generated callgraph will be misleading for call chains including functions from shared libs, as these are ignored. Instead, you will see spontaneous calls seemingly coming from nowhere. GProf has to estimate costs done in a call, as it only gets call counts. Suppose a function C called from two places A and B with cost spent in C when called from A is 99%, and when called from B is 1%. GProf only gets the call count of 1 each, so it estimates a distribution of 50% to 50%.

A tool recently available for Linux is a system-wide flat profiler called OProfile [6]. It will be included in the 2.6 kernel, and patches exist for 2.4. It does statistical sampling for the whole system by running in the kernel. Controlling the profiling process and getting collected data is done via a /proc interface, but user-level binaries are provided. As OProfile can use hardware performance counters, it seems to be a very good replacement for the simple wall-clock timer sampling with standard GProf. Ideas for OProfile come from the performance tool suite DCPI for Alpha processors [7].

If you only want to get a rough idea of how many events happen over a full program run, you can use the PerfCtr [?] kernel patch together with PAPI [12], PCL [16] or Rabbit [15]. These tools allow reading of hardware performance counters inside of a given process. Tools for binary rewriting of executables and libraries like Pixie and ATOM seem not to be available for Linux. Paradyn [9] allows for insertion of custom code snippets into even running processes.

The use of Valgrind as profiling tool will be explained in the next chapter. Some basics of Valgrind and features of Calltree, an extension of Cachegrind, are explained.

A commercial tool available for Linux is Intel VTune for X86 processors, both for data aggregation and visualisation. A profile visualisation with a Web frontend is HPCView [10]. "prophIt" is a profile visualisation for JAVA profiles, using a nice 3D view of a tree map (quite similar to KCachegrind, see below).

For a good general summary on Linux development tools, see [17].

3.2 Simulation with Runtime Instrumentation

Calltree uses the Valgrind supervision framework to control and trace a given binary, the target executable. The concept found in Valgrind is runtime instrumentation via binary translation. For some understanding of the benefits and limits of this approach, we start with a short description of Valgrind internals.

3.2.1 Valgrind Internals

Valgrind is a X86 CPU emulator. For a given set of target code, it analyses this code by using knowledge of the X86 instruction set. This results in a translation of the code into an intermediate, virtual RISC instruction set which is to be instrumented in an arbitrary way. The instrumentation phase itself is largely simplified by using the RISC set. Afterwards, retranslation into X86 code is done. These translation steps are done on demand for small code ranges called basic blocks. A basic block is a sequence of instructions with a control transfer at the end. After translating a given basic block, the translation is stored into a cache for later reusage and is accessible via a hash table, using the starting address of the original basic block as key.

When code is subject to translation and instrumentation before actually being executed, it is said to be running on the simulated CPU. All target code is to be run this way. In contrast, code of the Valgrind core itself always runs on the real CPU. Conceptually, the simulated and the real CPU will be running simultaneously; thus, it never should be possible that the same code runs on both CPUs, as there's no locking done for concurrent data accesses. For example, this means that the C runtime library is part of the target code and thus, not available to be called in Valgrind core.

Startup

Valgrind does load itself into the address space of the process to be supervised as a shared library. There is a script provided called "valgrind", which puts Valgrind command line arguments into an environment variable, sets LD_PRELOAD to the Valgrind shared library, and starts the target executable. The Valgrind library will be loaded and initialized first. Per design, the runtime loader is part of the target code, but when loading Valgrind itself, it still runs on the real CPU. This is the one and only exception to rule given above. After some initialisation, Valgrind switches to the simulated CPU as early as possible, before loading of other libraries and starting up the target code. I.e. the real CPU will always be stuck in the initialisation routine of the Valgrind library.

Main Loop

Switching to the simulated CPU means executing of the main loop of Valgrind. Given the start address of the next basic block of target code to be executed, the main loop does the following steps:

1. Check if there already exists an instrumented translation for the next basic block to execute by looking up in the translation hash table.
2. If no, trigger translation of this basic block.
3. Execute the translated code. It always jumps back into the Valgrind main loop at the end, and specifies the address of the next basic block. Depending on some return value, system calls can happen, or if the target program terminates, Valgrind will break out from the loop.
4. Every few executions of basic blocks, check for signal handlers to be run and jump to the thread scheduler (provided by Valgrind's own PThread implementation).
5. Go to 1.

The target code is able to trigger special Valgrind actions by using a special NOP sequence of X86 code. This sequence, called "client request", will be detected by the translation engine, and a call to skin or core provided code will be inserted into the translation. Thus, it's kind of a control transfer from simulated to real CPU. This way, Valgrind provides some replacements for C runtime functions or its PThread implementation.

3.2.2 Instrumentation for Cache Simulation

For cache simulation, every memory access of the target code has to be trapped. This is done by inserting calls to the provided cache simulator by instrumentation. For every basic block, the instrumentation function also allocates space for event counters per instruction. At program termination, these counters are written as ASCII to a file called "cachegrind.out.<processID>". This ASCII file contains the data already related to debug information. I.e. instead of basic block addresses, one gets function names and cost events related to source code lines.

The simulator can emulates an Level1 and Level2 cache; it determines cache parameters to be simulated via CPUID, i.e. it simulates the cache of the machine cachegrind runs on. With multithreaded target code, the simulator doesn't distinguish threads. Thus, memory accesses of all threads are simulated to be routed over the same cache hierarchy.

Interestingly,[14] has a comparision of Cachegrind generated results and real results got by the tool Rabbit [15]. Depending on the application, Cachegrinds results are quite near to reality.

3.2.3 Call Tracing

There are some extensions done in Calltree. We first touch the differences regarding cache simulation:

- Calltree allows for event counters to be accumulated separate per thread. This is done when using the option "–dump-threads=yes". To the filename of the dump files, "-<thread>" is appended.
- This is extended by allowing event counters to be accumulated depending on the context of the function currently active. This includes separation by recursion level and a number of callers in the current call chain. In the dump, the context is appended to a function name using an apostrophe as separation. Thus, effectively different functions seem to be executed. For example, the name for a function A to be separated by at most 2 recursion levels and the caller, and currently running at 5nd recursion level, called via X > Y > A will be "A'2'Y".
- Calltree allows to switch event accumulation on/off, create dumps, or zero event counters while running. These actions can happen when entering/leaving a function with a given name, or - in an interactive way - by creating a file "cachegrind.cmd" in the working directory of a Calltree run, consisting of a command that specifies the action to happen.

For the first two points mentioned, we need the possibility to store multiple event counters for the same basic block. Thus, event counter addresses to be incremented can change between multiple executions. To allow for this, the instrumentation adds a call to a setup function at start of every basic block. This adjust a start adress for an event counter area of this basic block execution, and the original calls to the cache simulator only get offset for possible incrementation of counters.

The first main task of Calltree is to keep track of the current active function with a call chain context and recursion level. To detect recursion, it stores the "active" status of functions. The second main task is to recognize CALL and RET instructions, and adjust call arc counters accessable via a hash table and keyed by the caller/called function pair. CALL and RET instructions can be trapped in the above mentioned setup function: For this, instrumentation was also changed to store the kind of control transfer of the last basic block execution into a global variable. For fast access to arc counters on RET, a stack of pointers of active arcs is used. This corresponds to the stack of current active functions. For each stacked arc, the real stack pointer at call time is stored. This allows the stack to be correctly unwinded when the real stack pointer is changed in a longjmp function or for exception handling, i.e. when CALL and RET instructions don't pair.

The last fact shows that Calltree can't cope with arbitrary stack switches in target code. An exception is multithreaded code, because Valgrind allows for trapping thread switches: Calltree switches to a different arc stack on a thread switch. Another exception are signal handlers which can have its own stack. This is handled in a similar way.

The ASCII dump format of Calltree is upwards compatible to Cachegrinds format. I.e. every visualisation tool that can load Calltree dumps will also be able to load Cachegrind dumps. Added for call tracing is the notion of a call happening at a source line to another function. For each call, the events happening while inside of the call are written out, too. The format was also extended to allow to write event counters on instruction granularity, as this information is already collected by Cachegrind itself. This allows for a visualisation tool to show annotated assembler by using the disassembler of "objdump" (part of LINUX "binutils").

By using "cachegrind.cmd" requests already mentioned above, one can request information on the current status of a Calltree run. Information, for example the current stack trace, is written to "cachegrind.res". This can be used for visualisation to give regular feedback on the status of the profile run.

We finish this chapter with a few numbers on runtime. Table 2 shows a comparision of the real time of a program run with the time it takes to run under Valgrind supervision without skin, with Cachegrind, and with Calltree. Calltree can either run with cache simulation switched on or off. One always gets instruction counts and call relationship, which should be enough for a lot of use cases. It should be noted that despite of the slowdown factors, GUI applications are quite usable on modern machines.

4 Meaningful Visualisation

When thinking about optimizing your code, questions the profiling process should answer are:

1. Which part of one programmers code is responsible for most of the time spent in the program? The answer is sometimes difficult when the real time is spent in some 3rd-party library code. Thus, one has to go back the call chain until one reaches own written code. This strategy goes wrong when the time spent in the 3rd-party library depends on parameters previously set.

2. Why does a given code take that many time? The number of times a function is called is important here, as much as the number of times loop bodies are executed inside of functions.

For visualisation issues, one both needs the big picture for the whole program and details, often down to costs spent for an instruction. A side effect of the ability to show smallest details is trust-worthiness into the profiling tool, because often, cost numbers for single instructions can be understandable: if the developer can't trust a tool, he won't use it. Also, browsing performance results has to be easy and quick, and visualisation has to be quickly recognizable.

4.1 KCachegrind Features

As the author already had C++ and KDE/QT programming experience, this platform was choosen for the visualisation tool. The screenshots in fig. 4 and fig. 5 show the current development version.

Command	Real	Valgrind with skin none	Cachegrind	Calltree	Ct. w/o sim.
bzip2 libc.so.6	1.70s	5.3s	44.5s	78.4s	26.1s
cc1 ct_main-i.c	0.72s	6.5s	46.9s	98.3s	41.4s
qtconfig	0.38s	6.1s	32.6s	45.4s	21.9s
konqueror /lib	1.21s	11.3s	63.7s	80.5s	39.3s

Table 2: Real runtime of program runs compared to execution with Valgrind skins.

KCachegrind tries to fit the above discussed requirements by providing the following features:

- Tool Indepence. It allows for visualisation of any kind of event types, and virtual event types can be calculated from real ones (Cost Types View).
- Quick browsing up/down call chains in the call graph consisting only of calls with high costs (Call Graph View), but still provides call details (Call/Caller Lists).
- Quickly recognizable performance problems even deep down in the call hierarchy (Tree Map View), and a sum list for cost of all functions reachable from a given one (All Call/Caller Lists).
- Overview on the high level (Function Grouping), down to annotated Source and annotated Assembler with jump visualisation.
- Some kind of time line mode when loading multiple profiles of one run, but captured in different time intervals (not shown).

From the above feature list, the callgraph browser and the annotated assembler view will be in the next release of KCachegrind. Still, features missing for a 1.0 release are

- Comparation Mode. This is crucial for quick checking the result of an optimization (Before/After). At the moment, this is a difficult task as it's only possible by comparing two profiles loaded in two windows.
- Controlling the profiling tool via GUI. This will allow runtime inspection of existing profile runs.

Another feature that seems to be needed is the addition of a GProf import filter. Even if this tool currently is very limited, it's quite possible that it will be enhanced by usage of the OProfile facility in the Linux kernel. Advantages over CallTree are more exact event counters by using hardware performance counters, system-wide profiling, and original run time in contrast to the slowdown incured by runtime instrumentation and cache simulation. An import filter for JAVA profiles (got with "java -prof ...") seems feasable.

4.2 A Profiling Session

In this section, we will sketch a typical performance analysis procedure. Browsing in unknown source with KCachegrind can be very interesting, but also time consuming. One should concentrate on a few main issues and stop when all assumptions on the performance of ones code are meet.

How to start after loading the profile dump?

- Function "main" should be selected. Look at the Call Graph and check if the time partitioning seems to be right going top/down.
- Look at the sorted function List on the left and click on the header of the column "Self". This sorts functions according self/exclusive costs. Look for the function with highest cost. If you find functions you don't have expected here, use the Call Graph and check the call chain to these functions, until you reach code of yourself. Check for optimization possiblites, by using the Source/Assembler view.
- Repeat the last step with the Call Count Column. Often, some call counts are higher than expected.

For understanding cost partitioning, a parallel view of the Call Graph, Call Tree Map and the "All Calls" View together with selecting functions in these views is useful, because selections

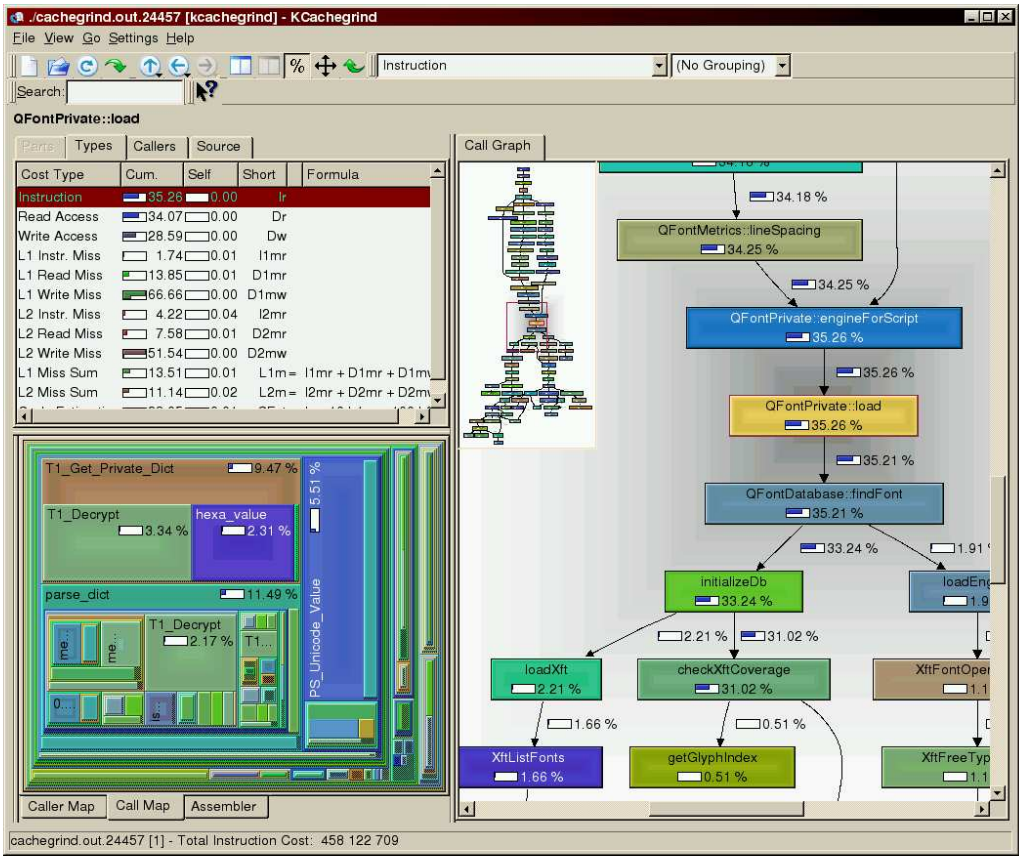

Figure 4: Cost Types (Top), Call Graph (Right) and Call TreeMap (Bottom)

in one view will change the selection in the other views, too.

For better understanding, the following profile sessions should be checked out by the reader.

The secrets of "xclock"

This will show that profiling of code without any debug info is possible, too. You will easily see that the xclock from XFree 4.3 spends most of its time in the XFT library. Did you have expected this?

Konqueror Startup

This shows the problematic side of simple profiling approaches. When cycle detection is switched on, everything is put into a cycle. How to make any use of this?

HTML Rendering

Try to trigger multiple profile dumps be using "echo D > cachegrind.cmd". Get a profile for loading of a large HTML page only. Is the result the same as you have expected?

JavaScript - "Useless profiling"?

Load a site with JavaScript. Try to detect the functions of the JavaScript Interpreter which are executed most of the time when running JavaScript. What are the most obvious problems here? (Hint: Scope searches for variables seem to be done very often).

How could we do profiling of the JavaScript Code? (Hint: We at least need profile values related to JavaScript Source Lines. We could introduce some client request for Calltree and inserting these into the JavaScript Interpreter. This way, Calltree will know for which JavaScript source line events happened. But this is future talk).

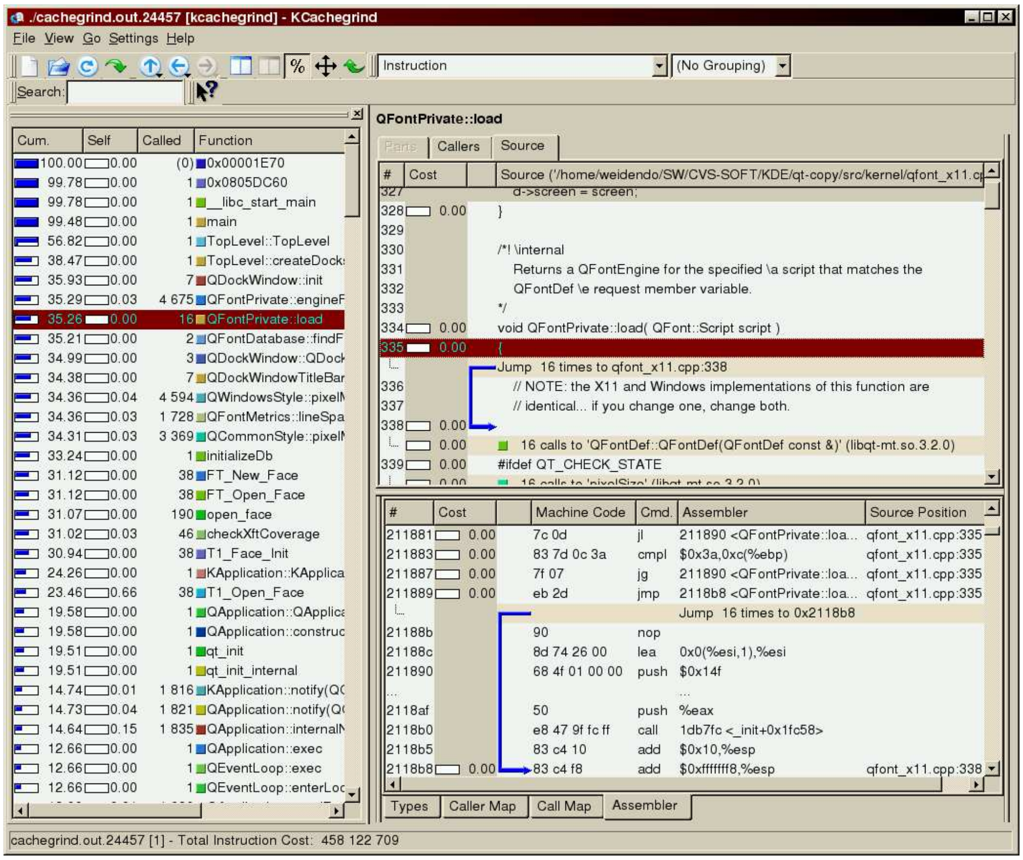

Figure 5: Sorted Function List (Left), Annotated Source (Top) and corresponding Assembler (Bottom)

5 Summary and future directions

In this paper, two tools were presented, helping the application developer in optimizing his programs. Hopefully they will convince many developers in the open-source community to optimize their code. Chances are quite good, as performance optimization has been recognized as being important for application acceptance by end users. But the top argument should be Calltree/KCachegrind's easy and comfortable usage. In contrast to GProf, needed instrumentation is done at runtime even for shared libraries and dynamically loaded plugins; profiling doesn't need any preparation.

There is quite some work to be done for the tool suite to reach a 1.0 status, though. Most needed is a mode allowing to compare two profiles. Controlling the profiling tool via GUI could be attractive with immediate feedback on the running status.

Relating data access events to data structures, and not only to instructions triggering these accesses is a issue for future research. It would help one to see if a given data structure layout needs to be changed to be e.g. more cache-friendly.

Acknowledgement

I deeply would like to thank Julian Seward for the wonderful tool Valgrind, and Nicolas Nethercote for his cache simulator Cachegrind, being the base that Calltree builds on. Without these tools, KCachegrind never would have gone into existance. I also want to thank the KDE community for providing the desktop I use every day. May the tools presented here be of some help for optimizing code in this project.

References

[1] J. Seward and N. Nethercode. Valgrind, an open-source memory debugger for x86-GNU/Linux. Homepage. http://developer.kde.org/~sewardj/.

[2] J. Weidendorfer. KCachegrind, a profiling visualisation tool. Homepage. http://kcachegrind.sourceforge.net.

[3] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. Proceedings of the Third Workshop on Runtime Verification (RV'03), Boulder, Colorado, USA, July 2003. To appear.

[4] G. Ammons, T. Ball, and J.R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In Proceedings of PLDI '97, June 1997.

[5] S. Graham, P. Kessler and M. McKusick. GPROF: A Call Graph Execution Profiler. In Proceedings of the SIGPLAN'82. Szmposium on Compiler Construction, SIGPLAN Notices, 17(6). pp. 120-126, June 1982.

[6] J. Levon. OProfile, a system-wide profiler for Linux systems. Homepage. http://oprofile.sourceforge.net.

[7] J.M. Anderson, L.M. Berc, J. Dean, et al. Continuous Profiling: Where Have All the Cycles Gone? ACM Transactions on Computer Systems, 15(4), pp. 357-390, November 1997.

[8] B. Cmelik and D. Keppel. Shade: A Fast Instruction Set Simulator for Execution Profiling. SIGMETRICS, Nashville, TN, 1994.

[9] B.P.Miller, M.D. Callaghan, J.M. Cargille, et al. The Paradyn Parallel Performance Measurement Tool. IEEE Computer 28(11), pp. 37-46, November 1995.

[10] J. Mellor-Crummey, R. Fowler, D. Whalley. Tools for Application-Oriented Performance Tuning. In Proceedings of 15th ACM International Conference on Supercomputing, Italy, June 2001.

[11] O. Traub, S. Schechter, and M.D. Smith. Ephemeral Instrumentation for Lightweight Program Profiling. In Proceedings of PLDI '00, 2000.

[12] K. London, J. Dongarra, S. Moore et al. End-user Tools for Application Performance Analysis, Using Hardware Counters. Presented at International Conference on Parallel and Distributed Computing Systems August 2001.

[13] M. Pettersson. X86 Performance counters driver for Linux. http://www.csd.uu.se/~mikpe/linux/perfctr/.

[14] N. Nethercote and A. Mycroft. The Cache Behaviour of Large Lazy Functional Programs on Stock Hardware. Proceedings of the ACM SIGPLAN Workshop on Memory System Performance (MSP 2002), Berlin, Germany, July 2002.

[15] D. Heller. Rabbit. A Performance Counters Library for Intel/AMD Processors and Linux. Homepage. http://www.scl.ameslab.gov/Projects/Rabbit.

[16] R. Berrendorf and B. Mohr. PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors. Version 2.2. Online PCL documentation. Homepage: http://www.fz-juelich.de/zam/PCL.

[17] G. Milmeister. Linux Development Tools. http://www.hotfeet.ch/~gemi/LDT.