NIST Technical Note 1790

Configuration of profiling tools for C/C++ applications under 64-bit Linux

David Flater

http://dx.doi.org/10.6028/NIST.TN.1790

National Institute of Standards and Technology
U.S. Department of Commerce

NIST Technical Note 1790

Configuration of profiling tools for C/C++ applications under 64-bit Linux

David Flater
Software and Systems Division
Information Technology Laboratory

http://dx.doi.org/10.6028/NIST.TN.1790

March 2013

U.S. Department of Commerce
Rebecca Blank, Acting Secretary

National Institute of Standards and Technology
Patrick D. Gallagher, Under Secretary of Commerce for Standards and Technology and Director

National Institute of Standards and Technology Technical Note 1790
Natl. Inst. Stand. Technol. Tech. Note 1790, 20 pages (March 2013)
http://dx.doi.org/10.6028/NIST.TN.1790
CODEN: NTNOEF

Configuration of profiling tools for C/C++ applications under 64-bit Linux*

David Flater <dflater@nist.gov>

March 2013

Abstract

Application profiling tools are the instruments used to measure software performance at the function and application levels. Without careful configuration of the tools and the environment, invalid results are readily obtained. The errors may not become obvious if a large, complex application is profiled before more simple validations are attempted. A set of four simple, synthetic reference applications was used to validate configurations for profiling under x86_64 Linux. Results from one validated configuration and examples of observed invalid results are presented. While validation results for specific versions of software quickly lose value, this exercise demonstrates how future configurations can be validated and shows the kinds of errors that may reoccur.

1 Background

Although application profiling tools can provide a range of different kinds of measurements, the most powerful measurement method available and the focus of this report is sampling-based profiling, where an application is interrupted based on some event to collect data on what it was doing when the interrupt occurred. Using hardware interrupts that are enabled by a separate profiling tool and then serviced by a kernel subsystem, it is possible to collect data about the inner workings of an application without modifying its source code or executables.

The simplest form of analysis for the resulting data simply ranks the functions of an application by the proportion of samples in which they were found to be currently executing. This proportion is called the *self time* of a function because it represents only the time spent executing the function itself, not the time spent executing any subfunctions that it invokes.

Those functions that directly consume the most CPU time are readily identified by self time. However, self time alone provides no information on why those functions were called. Profiling tools therefore support the collection of call chains, which are the hierarchical sequences of function calls leading down to the invocation of a given function. From call chains, the *total time* of functions, which includes both the self time and the time spent executing any subfunctions, can be determined.

When a sample is taken, there are three different methods that a profiling tool can use to determine the call chain of the currently executing function:

1. The call chain can be determined using static call frame information that recent compilers will include in the DWARF debugging data [1] that are typically included in the executable. This is the least invasive method.

*Specific computer hardware and software products are identified in this report to support reproducibility of results. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the products identified are necessarily the best available for the purpose.

2. The call chain can be determined using frame pointers, a legacy x86 run-time feature/protocol that is by default not done on x86_64, but that can be enabled with a compiler switch. Enabling frame pointers ties up one CPU register and produces extra code that pushes this register on the stack at the beginning of each function and pops it off the stack at the end of each function. Since the frame pointer protocol incurs a small but measurable run-time overhead [2], this method is slightly more invasive.

3. Finally, a profiling tool that instruments (modifies) programs to add its own data collection infrastructure can use that arbitrary instrumentation to keep track of call chains as functions are called and exited and simply retrieve that information when needed. This is the most invasive method.

There are a great many tools for application profiling. However, the selection of a particular platform, programming language, compiler or interpreter can quickly limit the choices. This report covers the following:

- Although various versions of the Linux kernel were tested, only x86_64 Linux was used.
- Perf [3], a high-powered profiling toolset, is included in the Linux kernel source tree and therefore inherits its versioning from the kernel. The ability to use DWARF debugging data instead of frame pointers to determine call chains first appeared in version 3.7, which was declared stable on 2012-12-10.
- OProfile [4], a similar high-powered profiling toolset, includes a new data collection tool, operf, that is built on the same kernel subsystem as perf, as well as a "legacy" tool, opcontrol, that uses a different method to collect data. As of 2012-12-26, the current version is 0.9.8, released 2012-08-27. It does not yet support the use of DWARF debugging data.
- GNU gprof [5] is an older profiling tool that is integrated with the GNU Compiler Collection (GCC) [6]. GNU gprof is very similar to the earlier Berkeley gprof [7]. Unlike perf and OProfile, gprof inserts data collection instrumentation into applications when they are compiled. Data are collected only for objects that were compiled with that instrumentation, which typically excludes the kernel and requires the use of gprof-enabled versions of standard libraries if complete results are to be obtained.
- Gprof2Dot [8] is a visualization tool that reads the text-format reports from various profiling tools and produces call graphs in the DOT language supported by Graphviz [9]. Gprof2Dot r. 2012-11-25 implements a heuristic to infer the total time of functions given profiler reports that do not include those data. An improved (non-heuristic) calculation of total time for perf input has been implemented in a patch [10], for now called r. 2012-11-25-DWF.
- The only compiler used in this work was GCC. Different versions of GCC produce different implementations of frame pointers and DWARF debugging data in compiled executables; successful application profiling therefore requires control over the version of GCC that is used. As of 2012-12-26, the current version is 4.7.2, released 2012-09-20.

Because the relevant software evolves at a rapid pace, the validation results for specific versions will quickly lose value. Nevertheless, this exercise demonstrates how future configurations can be validated, and the errors observed with current configurations demonstrate the importance of validating future configurations before they are relied upon.

2 How to read the call graphs

Following sections of this report will refer to call graphs appearing in Appendix A. All of the graphs were generated by Gprof2Dot and follow the same conventions.

Functions are represented by square nodes and function calls are represented by arrows from the caller to the callee.

In most cases, the nodes contain the following four pieces of information, from top to bottom: the name of the executable object containing the function, the name of the function, the approximate total time of the

function, and the approximate self time of the function expressed as a percentage. If all reported functions are in the same object, the first field may be omitted.

Arrows are labelled with an estimate of the proportion of a callee's total time that is attributable to the indicated caller.

Finally, nodes and arrows are colorized using a simple "heat map" scale in which functions with a large proportion of total time tend toward red and functions with little total time tend toward blue.

3 Configuration and use details

3.1 Compiler options for building the application

The following information is based on the behavior of versions 4.6.3 and 4.7.2 of GCC under x86_64 Linux. Future versions may, of course, behave differently.

If call chains are to be determined using DWARF data, these data must be present in the executable. It is the default behavior of GCC to include them (even without the –g switch), so the only requirement is not to remove them using the strip tool or the –s switch of GCC.

If call chains are to be determined using frame pointers, then the application to be profiled must be compiled with the –fno-omit-frame-pointer switch to override the default behavior on x86_64, and it should be built only with GCC version 4.6.x. This version constraint is to avoid the errors described in Section 5.2 and Section 5.3.

If gprof is to be used, then the application to be profiled must be compiled and linked with the –pg switch, and optimization must be limited to –O0 or –O1 to obtain call graph data. The latter is not a documented requirement, but when tested, –O2 and –O3 somehow disabled gprof's instrumentation:

```
gprof: gmon.out file is missing call-graph data
```

For accurate attribution of time spent in kernel space or shared libraries, those objects must also be built with DWARF data, frame pointers or gprof instrumentation, as applicable.

Less straightforward than the previous requirements is the decision of whether or not to prevent inlining of functions and similar optimizations. Inlining avoids the overhead of a function invocation by including the code of the invoked function within the calling function. A similar optimization involves replacing function invocations with simple jumps when the function invocations occur in a predictable, linear sequence.

Inlining does not necessarily invalidate profiling results, but it routinely obfuscates them. If profiling is performed on applications that were compiled with these kinds of optimizations enabled, the reported call chains and attributions of time will, at best, reflect the structure of the program as it was implemented by the compiler rather than the structure of the source code as it was written. Depending on the goals of the profiling exercise, this may or may not be a useful result. At worst, the optimizations done by the compiler may contradict assumptions made by the profiling tool and lead to confusing or invalid results.

The relevant GCC options that were used in this work were no-inline and no-optimize-sibling-calls, applied selectively using function attributes as needed to obtain predictable results in valid profiles. The impact on profiling of omitting these options is shown by examples in Section A.4.

Disabling inlining for the entirety of a C++ program is inadvisable as it will significantly skew the performance of the Standard Template Library [11].

3.2 Sampling events and frequency

Both perf and OProfile allow the user to choose numerous different events to drive event-based sampling and to adjust the frequency with which samples are taken. When the chosen event is that a particular hardware timer reaches a specified count, the sampling frequency is derived from the count and the timer's frequency. If a different sort of event is chosen, such as a tracepoint for a specific kernel operation, sampling need not occur at any regular interval and may occur in bursts.

If the sampling frequency at any time becomes too high, data collection tools may lose samples or the overhead of profiling may skew or completely overwhelm the application under test. If it is too low, too few samples will be taken and the results will not be representative of actual program performance. The frequency must be tuned to account for the capabilities of the system and the needs of the measurement.

For the examples in this report, perf and OProfile were configured to sample at a count of 10^6 CPU cycles. This yielded a sampling frequency of approximately 3 kHz on the system tested. (Perf also offers the option to specify a desired sampling frequency directly and derive the count from that.)

```
perf record -e cpu-cycles -c 1000000 ...
operf -e CPU_CLK_UNHALTED:1000000 ...
opcontrol --event="CPU_CLK_UNHALTED:1000000" ...
```

As a consequence of sampling at regular intervals, functions that do not account for much of an application's CPU time may be entirely missed by data collection or may be sampled only occasionally, appearing and disappearing at random from the results of repeated data collection runs. This happens with the kernel's timer interrupt and its callees in the examples in Appendix A. To prevent call graphs from becoming cluttered with such ephemeral functions, gprof2dot implements an adjustable total time threshold, which defaults to 0.5 % for function nodes and 0.1 % for edges. Perf-report implements a similar threshold, but on self time, to filter the reported call chains. The suppressed results are not necessarily invalid; they are merely deemed insignificant.

Gprof tracks function calls with embedded instrumentation, so even functions that do not account for much of an application's CPU time are reliably detected. However, gprof determines self time by sampling at a fixed frequency of 100 Hz, so the functions in question won't necessarily have significant values for self time.

The variability of measurements that can result from too few samples is a consideration for any application profile. Methods for quantifying that variability are provided in a separate report [12].

If a single run of an application at the maximum reasonable sampling frequency yields insufficient data and therefore too much uncertainty in the results, it is often statistically valid to pool the data from multiple runs to achieve the same effect as a higher sampling frequency. Gprof includes support for this operation.

3.3 Handling data volume with perf 3.7

In the transition from perf 3.6 (using frame pointers) to 3.7 (using DWARF), the volume of data written for call chain profiling increased dramatically, resulting in overload-related data loss on the system tested:

```
Warning:
Processed 14551 events and lost 2 chunks!

Check IO/CPU overload!
```

If sufficient RAM is available, the most effective mitigation is to direct the perf data to a tmpfs directory. If disk storage must be used, the failures are reduced but not eliminated by adding –r 1 to the command line to run perf at realtime priority.

4 Validation approach

Different configurations were tested using a suite of four very simple test cases written in C. Brief descriptions of the test cases are included at the top of the relevant subsections in Appendix A. Source code is available on request or from the NIST web site [13].

The configuration-specific –fno-omit-frame-pointer, –pg, and –O*n* compiler command-line switches were determined by Makefile logic depending on the choice of profiling tool. Configuration-independent options to control inlining of test functions were specified in the source code of the test programs using function attributes:

```
__attribute__((noinline,optimize("no-optimize-sibling-calls"))) void fn2() {
```

These function attributes were altered only for Figure 11 and Figure 12 where they were disabled to demonstrate the effects.

Another optimization that would yield degenerate results is code elimination. A compiler will frequently delete variables whose values are not used and cascade this deletion onto the executable statements that generated those values. Any synthetic workload intended to do nothing except create a predictable amount of busy-work is vulnerable to being eliminated in this way. Two techniques were used to combat this. First, declaring variables with global visibility makes it difficult for the compiler to conclude that their values are never used or altered by external code. Second, printing their values at the end of the program ensures that they cannot be eliminated unless the compiler can determine their final values through static analysis. In test cases that take no external input, it is still theoretically possible for a compiler to optimize away the entire workload and substitute a printout of the final result, but these two techniques have proven effective thus far.

Each test case is parameterized to permit adjustment of the number of iterations to perform in busy-work loops. Val1 (the first test case) is further parameterized to permit adjustment of the number of leaf functions that the program should have. These parameters were altered from their default values only for Figure 3 and Figure 4 where val1 was tweaked to demonstrate anomalies that correlate with shorter execution times.

5 Catalog of observed errors

5.1 Type 1 missing caller (untimely sampling)

A call chain is reported that erroneously omits the actual caller of a function, making it appear as if it were invoked directly by the caller's caller. This particular anomaly is known and documented to appear when a sample is taken while the call frame is being constructed at the very beginning or destroyed at the very end of a function's invocation [14]. Although these anomalous call chains can be collected in any profiling run regardless of the application, some of the reporting tools by default implement a minimum time threshold to prune out low-frequency results, and this threshold usually prevents the anomalous call chains from being reported. However, if the frame construction/destruction overhead is significant in comparison to the execution time of an important function—as in "short" functions that are invoked frequently—then the anomaly causes significant skewing of the results that cannot be filtered out.

Because this anomaly is documented and endemic, the validation tests were designed to ensure that leaf functions have significant execution time. However, the anomaly can be reproduced by adjusting the parameters of the val1 test to do the opposite. Figure 3 shows the problem with results obtained by defining –DINNERLOOP=1; compare its call graph with the valid one in Figure 1.

The type 1 missing caller error is avoided by using DWARF or gprof instrumentation for call chains.

Figure 3 also reflects significant skewing of the reported self time of function1 relative to the expected results. This separate anomaly will be discussed in Section 6.

5.2 Type 2 missing caller (deferred frame pointer push)

A change made in GCC 4.7 allowed optimization to reschedule and defer the push of the frame pointer that previously occurred in the function prologue whenever frame pointers were enabled [15]. When binaries compiled with GCC 4.7.x are profiled using frame pointers, incorrect call chains are derived whenever a sample is taken between the top of the function and the instruction that pushes the frame pointer. In Figure 13, this resulted in the main program being erroneously omitted from half of the call chains, with collateral damage to the calculation of total time for main. For comparison, valid results are shown in Figure 10.

The error is avoidable by reverting to GCC 4.6 or by using DWARF for call chains.

5.3 Type 3 missing caller (missing frame pointers)

A behavior present in GCC 4.4 and 4.5 caused optimization attributes of functions, such as the `optimize("no-optimize-sibling-calls")` example in Section 4, to implicitly reset global options provided on the command line, including –fno-omit-frame-pointer [16]. Those functions for which optimization attributes were specified could thus unexpectedly end up without frame pointers, while other functions in the program would still have them. The consequences are similar to those of the previous error, but more severe (Figure 9). For comparison, valid results are shown in Figure 7.

While they were useful in the validation test cases, optimization attributes of functions are seldom used in general applications. The error is avoidable by promoting the optimization options to the command line (applying the same choices to every function in the application) or by upgrading to GCC 4.6 or 4.7.

5.4 Spurious recursion

The operf tool of OProfile 0.9.8 reports cycles of recursion that do not exist (Figure 6). For comparison, valid results are shown in Figure 5. The error reproduced with GCC 4.6.3 for kernel versions ranging from 3.7.1 back to 3.0.57.

This error is confined to operf's reporting of call chains and does not affect self time results.

5.5 Inaccurate distribution of total time

Having been designed originally for use with gprof, gprof2dot.py r. 2012-11-25 relies on a graph-based heuristic to estimate total time for functions. This results in an inaccurate distribution of total time for val3 (Figure 8). For comparison, valid results are shown in Figure 7.

The input to gprof2dot.py from perf includes complete call chains for every sample, so it is not necessary to estimate total times heuristically. Instead, total time may be calculated as the proportion of samples for which a function either was executing or was found on the stack [17]. gprof2dot.py r. 2012-11-25-DWF was patched to implement this change to the calculation of total time when the input is from perf [10].

5.6 Excessive variability of self time

When calculating the self time of functions from data collected by opcontrol while call graph profiling is enabled and sample separation is off, OProfile uses only about 1 % the number of samples that ought to be available for that calculation, resulting in increased variability of results due to the small sample size. Figure 2 shows comparable results from two consecutive runs. The self time of function1 was calculated as the proportions 36/130 and 44/114 respectively, with correspondingly fewer samples for the other functions.

With over 15 000 samples having been collected in each run, the count for function1 should have been over 5000 given the construction of the reference application.

In the output of opreport, the expected numbers of samples do appear in the context of calls from the main program—just not in the summaries for the five functions themselves:

```
samples  %        symbol name
--------------------------------------------------------------------------------
  5004     100.000  main
36       27.6923  function1
  36       100.000  function1 [self]
--------------------------------------------------------------------------------
  3941     100.000  main
36       27.6923  function2
  36       100.000  function2 [self]
--------------------------------------------------------------------------------
  3032     100.000  main
32       24.6154  function3
  32       100.000  function3 [self]
--------------------------------------------------------------------------------
  2019     100.000  main
18       13.8462  function4
  18       100.000  function4 [self]
--------------------------------------------------------------------------------
  982      100.000  main
8         6.1538  function5
  8        100.000  function5 [self]
--------------------------------------------------------------------------------
0              0  main
  5004     33.4090  function1
  3941     26.3119  function2
  3032     20.2430  function3
  2019     13.4798  function4
  982       6.5563  function5
  0              0  main [self]
--------------------------------------------------------------------------------
```

This behavior was reproduced with GCC 4.6.3 for OProfile 0.9.8 and kernel versions ranging from 3.7.1 back to 3.0.57, and for OProfile 0.9.7 and kernel version 2.6.34.13.

The error goes away if call graph profiling is disabled with the –-callgraph=0 option *or* if opcontrol is started with the –-separate=kernel option. N.B., options are saved in /root/.oprofile/daemonrc and persist from run to run until explicitly changed.

While this error is confined to opcontrol, variability of measurements is a consideration for any application profile and is addressed in a separate report [12].

6 Anomalous results with short functions

Figure 3 reflects significant skewing of the reported self time of function1 relative to the expected results. Defining –DINNERLOOP=1 for val1 has the effect of reducing the number of busy-work iterations in the leaf functions from 256 to 1 and of increasing the number of main loop iterations by the same factor of 256 to maintain a similar total run time. The main program therefore could be expected to have a greater proportion of self time than it would have in the default configuration. Nevertheless, the leaf functions should still show a linear relationship in the way self time is allocated among them. That did not occur in

Figure 3; the self time of function1 falls significantly below a line fit through the other four functions' results. (See [12] for additional data collection and analysis showing that the deviation is statistically significant.)

A review of the disassembled executable revealed no obvious reason why function1 would perform significantly differently in this case. The anomaly occurs for perf 3.7 using DWARF, for perf 3.6 using frame pointers, and for operf using frame pointers. The results from gprof appear to be closer to expectations once variability is taken into account—but gprof's instrumentation overhead and limiting optimization to –O1 both would be expected to increase the execution time of the leaf functions, dampening the effect of setting INNERLOOP=1.

The realm of plausible explanations includes both measurement bias and actual differences in performance. Regarding the former, the OProfile manual warns that x86 hardware-specific latencies in the delivery of profiling interrupts can skew the results in certain cases [18]:

> The problem comes from the x86 hardware; when the counter overflows the IRQ is asserted but the hardware has features that can delay the NMI interrupt: x86 hardware is synchronous (i.e. cannot interrupt during an instruction); there is also a latency when the IRQ is asserted, and the multiple execution units and the out-of-order model of modern x86 CPUs also causes problems.

This class of phenomena is often referred to as "skid," metaphorically likening a CPU to a speeding car that cannot stop short enough to service an interrupt exactly where it occurred. The precise sampling features of newer CPUs (Intel PEBS [19, §18.4.4] or AMD IBS [20]) are intended to minimize skid; however, there remains a systematic, single-instruction skid which would not necessarily be compensated for by a profiling tool [21]. (With branches, the location of the previous instruction might be ambiguous, and the systematic error therefore difficult to correct.)

Perf supports the use of PEBS and IBS through a parameter called precise-level. Running perf at its maximum supported precise-level of 2 ("SAMPLE_IP requested to have 0 skid") did not change the self time relationships among the five leaf functions and main. It did, however, transfer about 3 % of samples to an "[unknown]" function and cause invalid call chains to be reported, as seen in Figure 4.

Another plausible explanation for the skewed proportions of self time is that the short functions have brought an actual performance feature of the microarchitecture, such as branch prediction, cache, or CPU pipeline scheduling efficiencies, to the forefront. Function1 would then be showing an *actual* performance difference related to its being the first of the five functions that is invoked after the loop iteration logic in the main program is executed and the only one for which the branch is always true.

Reversing the order of the conditional function invocations in the main program of val1 did not produce an anomalously low result for function5. Instead, the five functions showed the progression that was expected, and the program completed in slightly less time. Disassembly revealed that the implementation of the main program was more efficient for the reversed case. The experiment thus showed that different coding of the main program results in a redistribution of reported self time among the leaf functions, but did not rule out measurement bias as a possible explanation (i.e., it is premature to conclude that the observed difference comes from an actual change in performance).

Until further research identifies a root cause for this anomaly, it is advisable to watch for short, frequently-invoked functions in profile reports and use results with extra caution whenever they are found. The call counts provided by gprof are helpful in determining which functions could be problematic.

7 Conclusion

Invalid profile results can be caused by a number of factors from misuse of the tooling to obscure version dependencies. A failed validation is only the first step in the process that leads to identifying and correcting a problem in the failing configuration. Similarly, a successful validation is only the first step in the process of applying a particular toolset to accomplish some goal. In either case, validation testing replaces second-hand information on what *should* work with first-hand, empirical results on how particular toolsets actually function *in situ* and may avert harmful consequences for the user when these two differ *for any reason*.

References

[1] The DWARF debugging standard website, 2012. http://dwarfstd.org/.

[2] David Flater and William F. Guthrie. Screening for factors affecting application performance in profiling measurements. NIST Technical Note, to appear.

[3] Perf: Linux profiling with performance counters, 2012. https://perf.wiki.kernel.org/.

[4] OProfile, 2012. http://oprofile.sourceforge.net/.

[5] Jay Fenlason. GNU gprof, 1988. In GNU binutils, http://www.gnu.org/software/binutils.

[6] GCC, the GNU Compiler Collection, 2012. http://gcc.gnu.org/.

[7] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: a call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982. http://docs.freebsd.org/44doc/psd/18.gprof/paper.pdf.

[8] José Fonseca. Gprof2Dot: Convert profiling output to a dot graph, March 2012. http://code.google.com/p/jrfonseca/wiki/Gprof2Dot.

[9] Graphviz—graph visualization software, 2012. http://www.graphviz.org/.

[10] David Flater. [Enhancement] Calculate true total time with perf, December 2012. http://code.google.com/p/jrfonseca/issues/detail?id=77.

[11] José Fonseca. Which options should I pass to gcc when compiling for profiling? In Gprof2Dot: Convert profiling output to a dot graph, December 2012. http://code.google.com/p/jrfonseca/wiki/Gprof2Dot#Which_options_should_I_pass_to_gcc_when_compiling_for_profiling?

[12] David Flater and William F. Guthrie. Estimation of uncertainty in application profiles. NIST Technical Note, to appear.

[13] David Flater. Application profiling tool validation test suite for Linux. http://www.nist.gov/itl/ssd/cs/software-performance.cfm.

[14] John Levon et al. Interpreting call-graph profiles, in OProfile manual, 2012. http://oprofile.sourceforge.net/doc/interpreting-callgraph.html.

[15] David Flater. GCC Bug 55667 [4.7 regression] –O1 enables frame pointer push to move around on x86_64, December 2012. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=55667.

[16] Richard Sandiford. GCC Bug 38716, undocumented __attribute__((optimize)) behavior when the attribute specifies no optimisation level, January 2009. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=38716.

[17] Mike Dunlavey. Answer to "Alternatives to gprof", August 2012. http://stackoverflow.com/questions/1777556/alternatives-to-gprof#1779343, item 4 (the myth "that recursion is a tricky confusing issue").

[18] John Levon et al. Profiling interrupt latency, in OProfile manual, 2012. http://oprofile.sourceforge.net/doc/interpreting.html#irq-latency.

[19] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2, August 2012. http://download.intel.com/products/processor/manual/253669.pdf.

[20] Paul J. Drongowski. Instruction-Based Sampling: A new performance analysis technique for AMD family 10h processors, November 16, 2007. http://developer.amd.com/assets/AMD_IBS_paper_EN.pdf.

[21] Intel VTune Amplifier XE 2013 Release Notes, August 2012. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe-2013-release-notes. See §6, Issues and Limitations: Running time is attributed to a next instruction.

A Gallery of valid and invalid results

A.1 val1: triangular distribution of time among N functions

With default values of the parameters, the expected distribution of time (both self and total) is approximately

Function	Time
function1	5/15
function2	4/15
function3	3/15
function4	2/15
function5	1/15
main	0

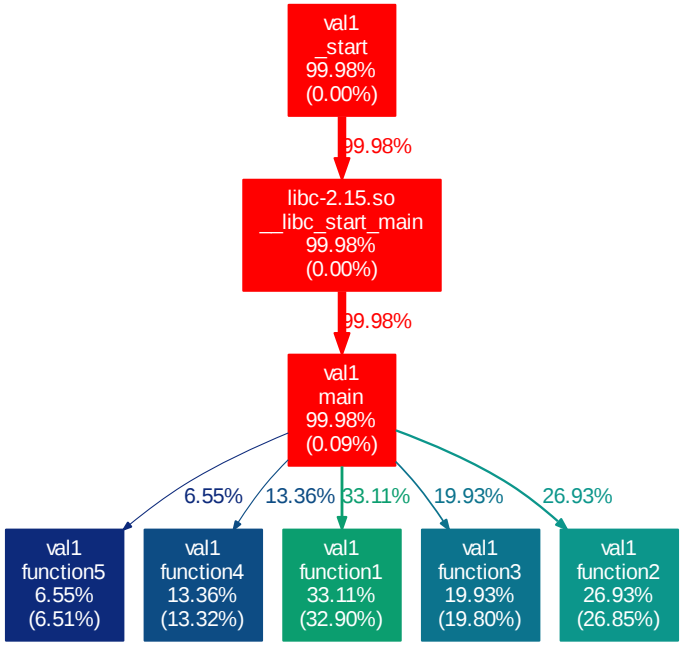

Figure 1: val1 valid result. — kernel/perf 3.7.1, GCC 4.7.2 (–O3), gprof2dot.py r. 2012-11-25-DWF

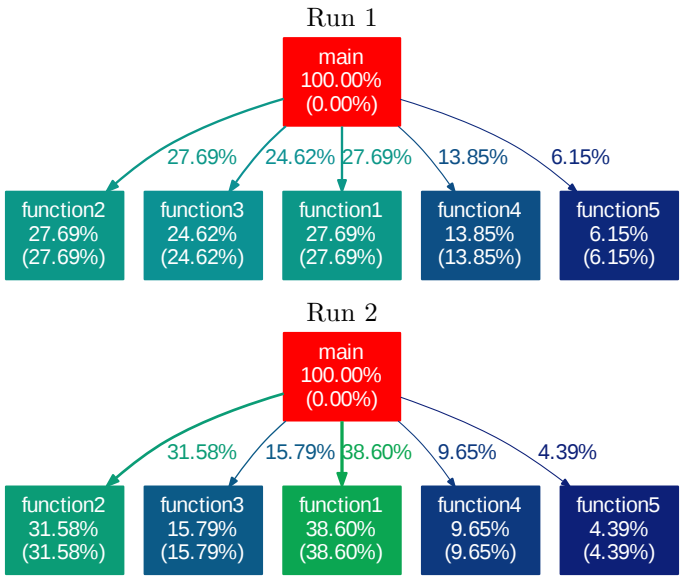

Figure 2: val1 excessive variability of self time. — kernel 3.7.1, OProfile (opcontrol) 0.9.8 (default –-separate=none behavior), GCC 4.6.3 (–O3 –fno-omit-frame-pointer), gprof2dot.py r. 2012-11-25-DWF

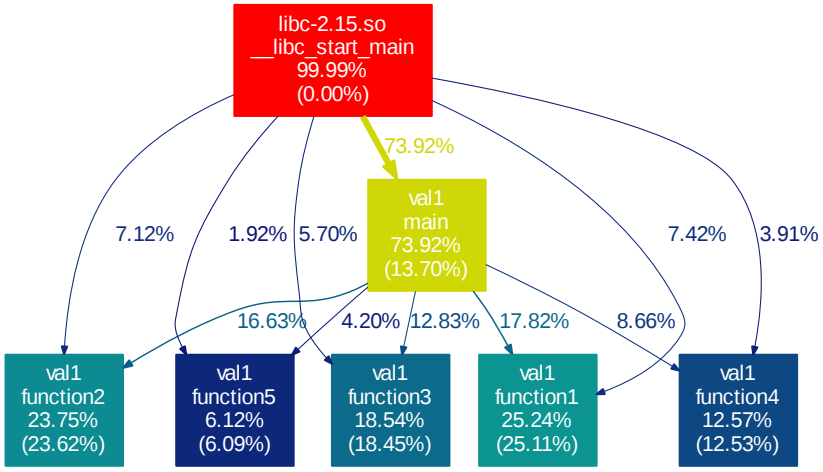

Figure 3: val1 (–DINNERLOOP=1) type 1 missing caller. — kernel/perf 3.6.11, GCC 4.6.3 (–O3 –fno-omit-frame-pointer), gprof2dot.py r. 2012-11-25-DWF

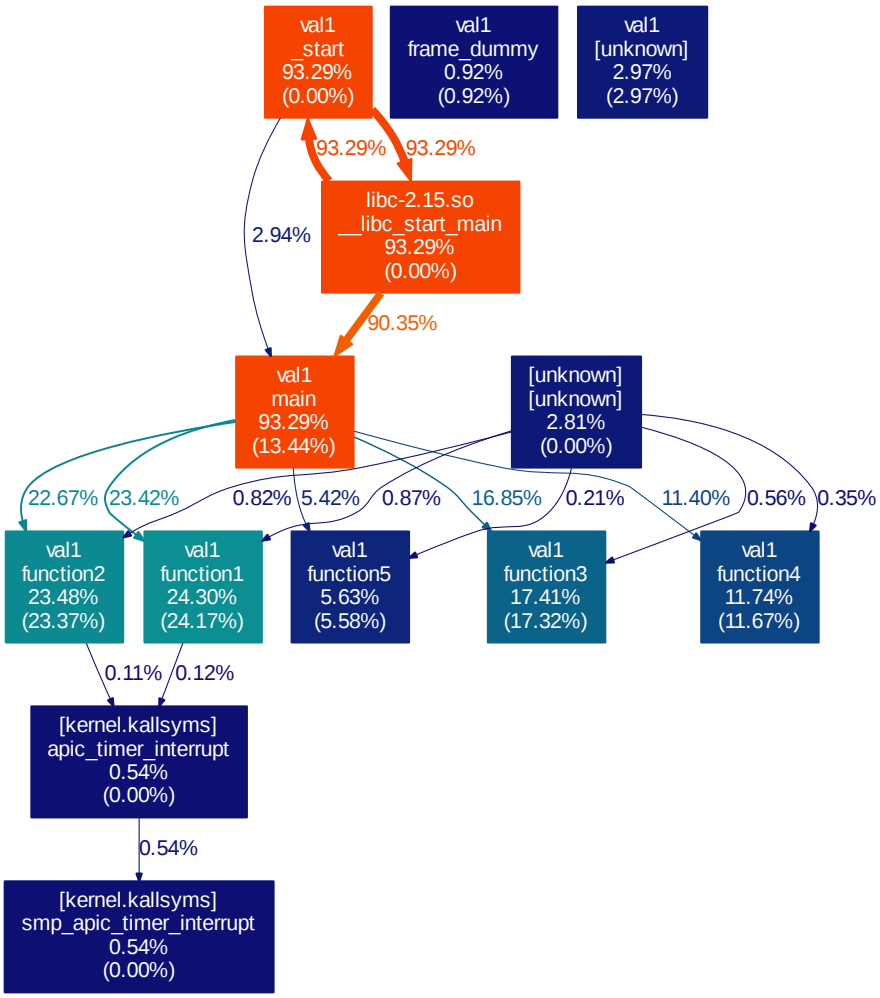

Figure 4: val1 (–DINNERLOOP=1) call chain anomalies with PEBS. — kernel/perf 3.7.1 (–e cpu-cycles:pp), GCC 4.7.2 (–O3), gprof2dot.py r. 2012-11-25-DWF

A.2 val2: two functions with a single invocation and no recursion

The call graph is simply
main→leaffn
with nearly 100 % of time spent in leaffn.

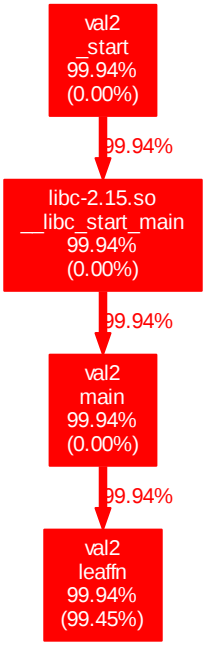

Figure 5: val2 valid result. — kernel/perf 3.7.1, GCC 4.7.2 (–O3), gprof2dot.py r. 2012-11-25-DWF

Figure 6: val2 spurious recursion. — kernel 3.7.1, OProfile (operf) 0.9.8, GCC 4.6.3 (–O3 –fno-omit-frame-pointer), gprof2dot.py r. 2012-11-25-DWF

A.3 val3: mutually recursive functions

With default values of FN1LOOP and FN2LOOP, the expected distribution of time is approximately

Function	Total time %	Self time %
main	100	0
cycfn2	100	67
cycfn1	67	33

resulting from the following call chains and proportions of samples:

Samples (scaled)	Call chain
2	main→cycfn2
1	main→cycfn2→cycfn1
2	main→cycfn2→cycfn1→cycfn2
1	main→cycfn2→cycfn1→cycfn2→cycfn1

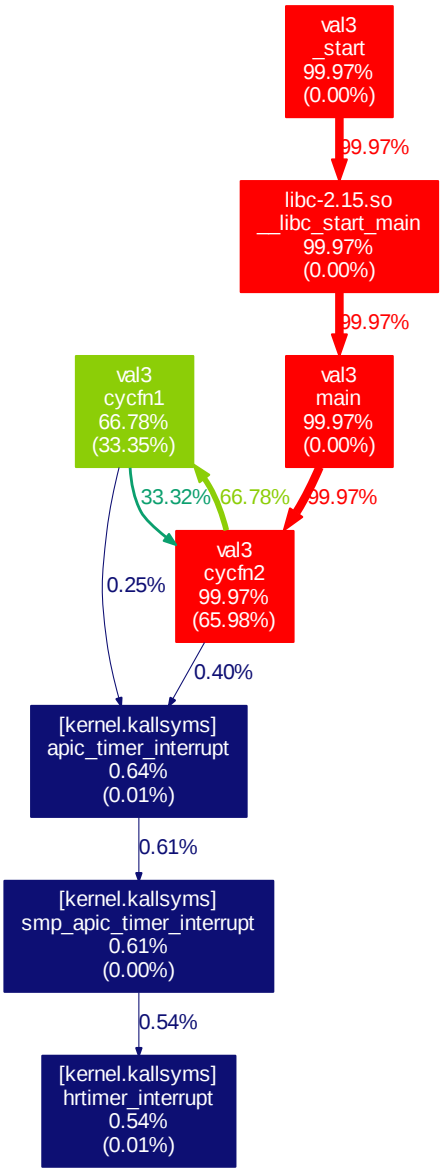

Figure 7: val3 valid result. — kernel/perf 3.7.1, GCC 4.7.2 (–O3), gprof2dot.py r. 2012-11-25-DWF

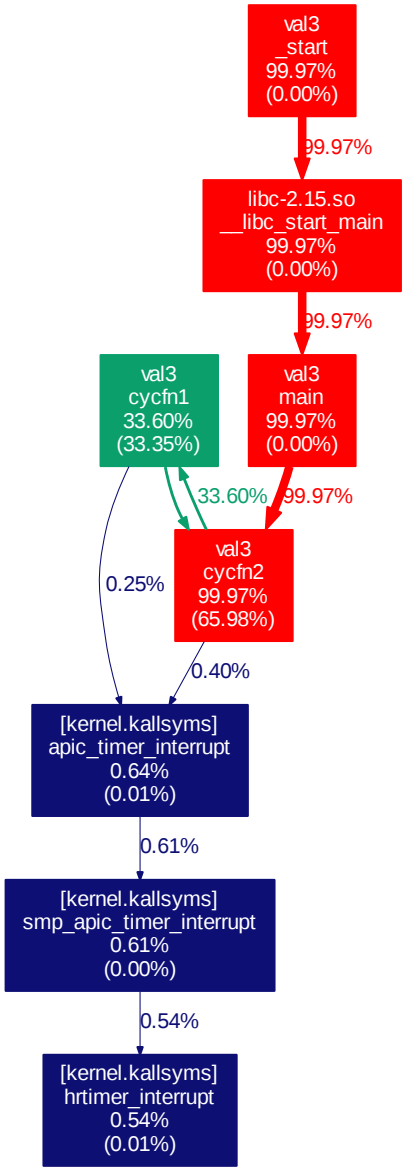

Figure 8: val3 inaccurate distribution of total time by heuristic in gprof2dot.py r. 2012-11-25. — kernel/perf 3.7.1, GCC 4.7.2 (–O3)

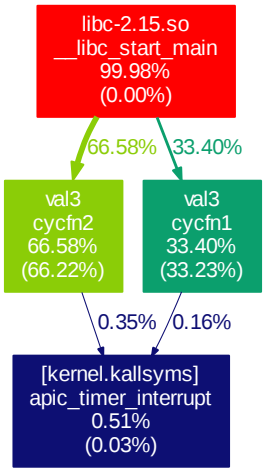

Figure 9: val3 type 3 missing caller. — kernel/perf 3.6.11, GCC 4.5.4 (–O3 –fno-omit-frame-pointer), gprof2dot.py r. 2012-11-25-DWF

A.4 val4: three functions with a single invocation and no recursion

With default values of FN1LOOP and FN2LOOP, the expected distribution of time is approximately

Function	Total time %	Self time %
main	100	0
fn2	100	50
fn1	50	50

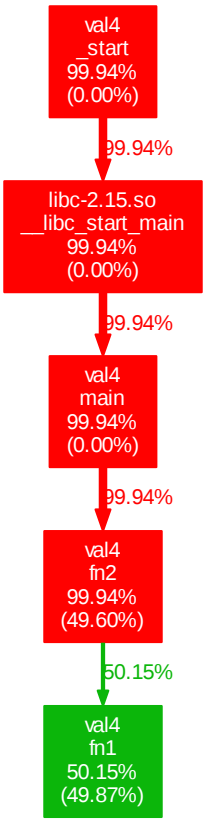

Figure 10: val4 valid result. — kernel/perf 3.7.1, GCC 4.7.2 (–O3), gprof2dot.py r. 2012-11-25-DWF

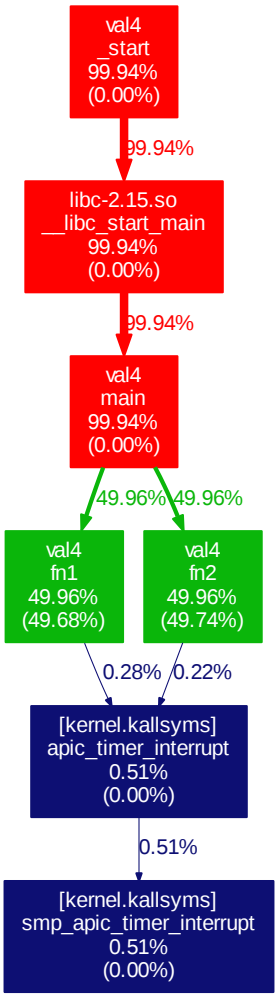

Figure 11: Id., no-optimize-sibling-calls attribute removed from source to demonstrate effect

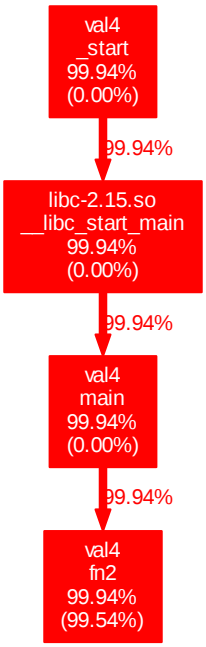

Figure 12: Id., no-inline attribute removed from source to demonstrate effect

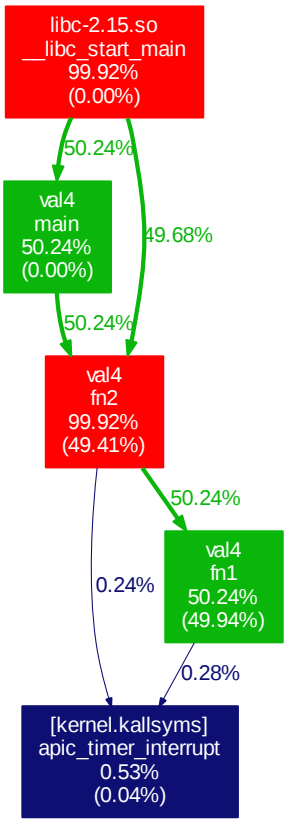

Figure 13: val4 type 2 missing caller. — kernel/perf 3.6.11, GCC 4.7.2 (–O3 –fno-omit-frame-pointer), gprof2dot.py r. 2012-11-25-DWF