perf file format

Student:
Urs Fässler

Supervisor:
Andrzej Nowak

CERN openlab
September 2, 2011

Abstract

Performance measurement of software under Linux is done with the perf system. Perf consists of kernel code and an userspace tool. The tool records the data to an file which can be analyzed later. Understanding this data format is necessary for individual software performance analysis.

This report provides information about the data structures used to read the data file. An application was written to demonstrate how the data file can be read. For a given data file, the application shows the frequency with which source code functions are used.

Contents

1 Introduction

In recent years, the speed of processors has not increased and the industry has moved towards parallel systems. The only way to increase calculation power is by adding more cores, but this creates higher demand for power and produces more heat. Another way is to take a closer look how our software works. This is exactly the point where we need performance measurement. Without having a clue where the bottleneck is, one does not know how to improve speed. [3]

For Linux, performance can be measured with the perf [4] system. It consists of some functionality inside the kernel and a userspace tool called perf. The tool is used to start the measurement in the kernel as also storing and displaying the data. This report will give a detailed description how the data file can be read and the information processed.

1.1 Performance counters

Performance counters are often realized as hardware counters. This has the advantages that it has a low overhead and also low perturbation since it does not use registers or the ALU. It is also widespread among different CPUs where it is often called a PMU (Performance Measurement Unit). The PMU can be programmed / configured by the user to count different kind of events. Examples for such events include executed cycles, branch misses and cache misses [1]. The basic structure of perf is shown on figure 1.

More information can be found on the level of the hardware [7], focusing on the Linux implementation [2], for an overview of perf [5] and a workshop which provide an deeper understanding of the PMU [6].

1.2 About this document

The information in this document was gathered with Linux version 2.6.39.3 (9.7.2011, git commit 75f7f9542a718896e1fbe0b5b6e8644c8710d16e). There is no guarantee that the information is valid for different versions. The focus is on x86 Systems. All the work was done on a computer with an Intel Core 2 Duo T7200 processor and Debian GNU/Linux operating system.

Different text styles are used to emphasis some content in the document, namely `code snippets`, `console commands` and *files*.

The following terms are used in the described meaning:

event a signal produced by the measurement unit, e.g. instruction counter

sample an measured occurrence of an event

record an entry in the data file, e.g. information about samples or meta information

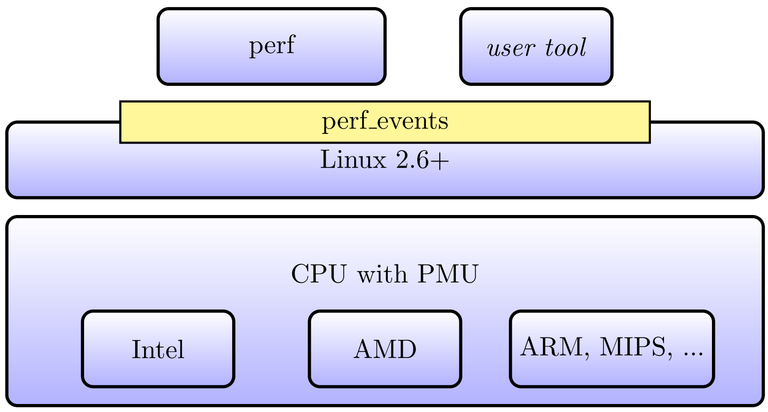

Figure 1: Overview of perf. It is based on the Linux kernel interface perf_events. The Linux kernel needs a CPU with an PMU to measure the hardware. It is also possible to write another performance measurement tool on top of the kernel interface.

2 The perf application

The perf application is part of the Linux kernel tools. The source code is found in the kernel sources in the directory *<linux source>/tools/perf/*. perf is comprised of several sub-tools for different tasks. These are for example the recording or reporting of events. Each of these sub-tools acts like an stand alone application, but uses a common infrastructure. The tools are executed with a command line argument for perf, e.g. `perf recording -h` or `perf report -h`.

2.1 perf record

The perf record tool is used to capture events and write them into a data file. By default, the data file has the name *perf.data* and is in the current working directory. It was used to capture all applications on all CPU's with timestamps. The command line to achieve this is `perf record -a -T`[1]. To capture on all CPU's, the pseudo file */proc/sys/kernel/perf_event_paranoid* has to have the content `0` or `-1`. This allows the kernel to use non-maskable interrupts which could cause an reboot of a running VirtualBox virtual machine.

During recording, several occurrences of an event are reported together. There exist two different modes. In the default case, the Kernel tries to measure 1000 samples per second. Therefore, it adjusts the sampling period dynamically [4]. With the switch `-c <n>`, a sample is generated for n events.

Figure 2 gives an overview how the recording works. First perf record initializes the recording via the perf_events interface of Linux. The records are then written into mmap pages[2] and a Linux signal is sent to perf record if a page is full. perf record then stores the records into the data file.

2.2 perf report

The perf report tool is for the analysis of the data file. By default it uses a text user interface where the usage of functions is shown. As an alternative, the information can be printed to *stdout*. With flags the focus can be changed. For example, `perf report -n -Caddr2line -i test.data` reads the file *test.data* and displays only samples for the application *addr2line*, but with the number of samples. Other filters are `-d` for dynamic shared objects and `-S` for symbols.

[1]But it seems that the `-T` flag has no influence on the recording

[2]not to confuse with the mmap record, they both have the same name

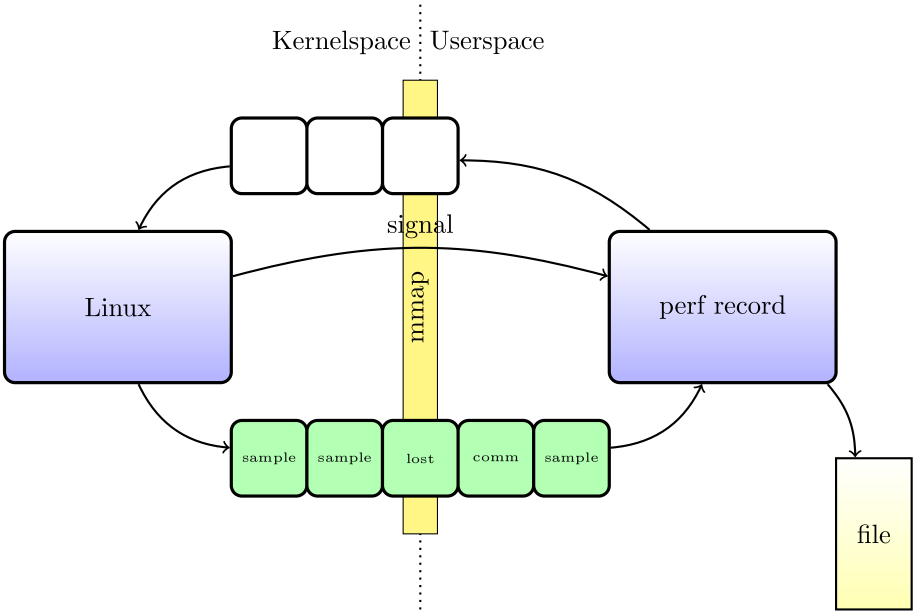

Figure 2: Operation of perf record. The kernel fills mmap pages with the records and send a signal if a page is full. perf record stores the records in the data file.

3 The perf file format

This section will give a detailed description of the perf file format. The file format is designed in such a way that it is upwards and downwards compatible. This is very convenient for the users, but makes the file format more complicated and therefore more difficult to understand. Nevertheless, the following description should give enough information to work with the perf data file.

In the tables describing the structures the convention for the data types is as following. `u<n>` is an unsigned integer with n bits. `char[<n>]` is a zero terminated string in a field with n bytes of memory. Another name in the type field refers to another structure.

3.1 Header

The perf data file header as shown in table 1 is at the beginning of the file. The `perf_file_section` structure is described in table 2. Figure 3 gives an overview of the connection between the structures and fields.

type	name	description
u64	magic	Magic number, has to be "PERFFILE".
u64	size	Size of this header.
u64	attr_size	Size of one attribute section, if it does not match, the entries may need to be swapped. We assume that it matches.
perf_file_section	attrs	List of `perf_file_attr` entries, see table 4.
perf_file_section	data	See section 3.2.
perf_file_section	event_types	List of `perf_trace_event_type` entries, see table 3.
u256	features	Unknown bitfield.

Table 1: `perf_file_header` from <*perf source*>*/util/header.h*

type	name	description
u64	offset	File offset of the section.
u64	size	Size of the section. If size is greater than the struct in the section, mostly this means that there are more than one structure of this type in that section.

Table 2: `perf_file_section` from <*perf source*>*/util/header.h*

type	name	description
u64	event_id	This entry belongs to the `perf_event_attr` entry where `.config` has the same value as this id. See table 5.
char[64]	name	Name of the event source.

Table 3: `perf_trace_event_type` from <*perf source*>*/util/event.h*

type	name	description
perf_event_attr	attr	see table 5
perf_file_section	ids	list of u64 identifier for matching with `.id` of the perf sample, see table 10 and 11

Table 4: `perf_file_attr` from <*perf source*>*/util/header.c*

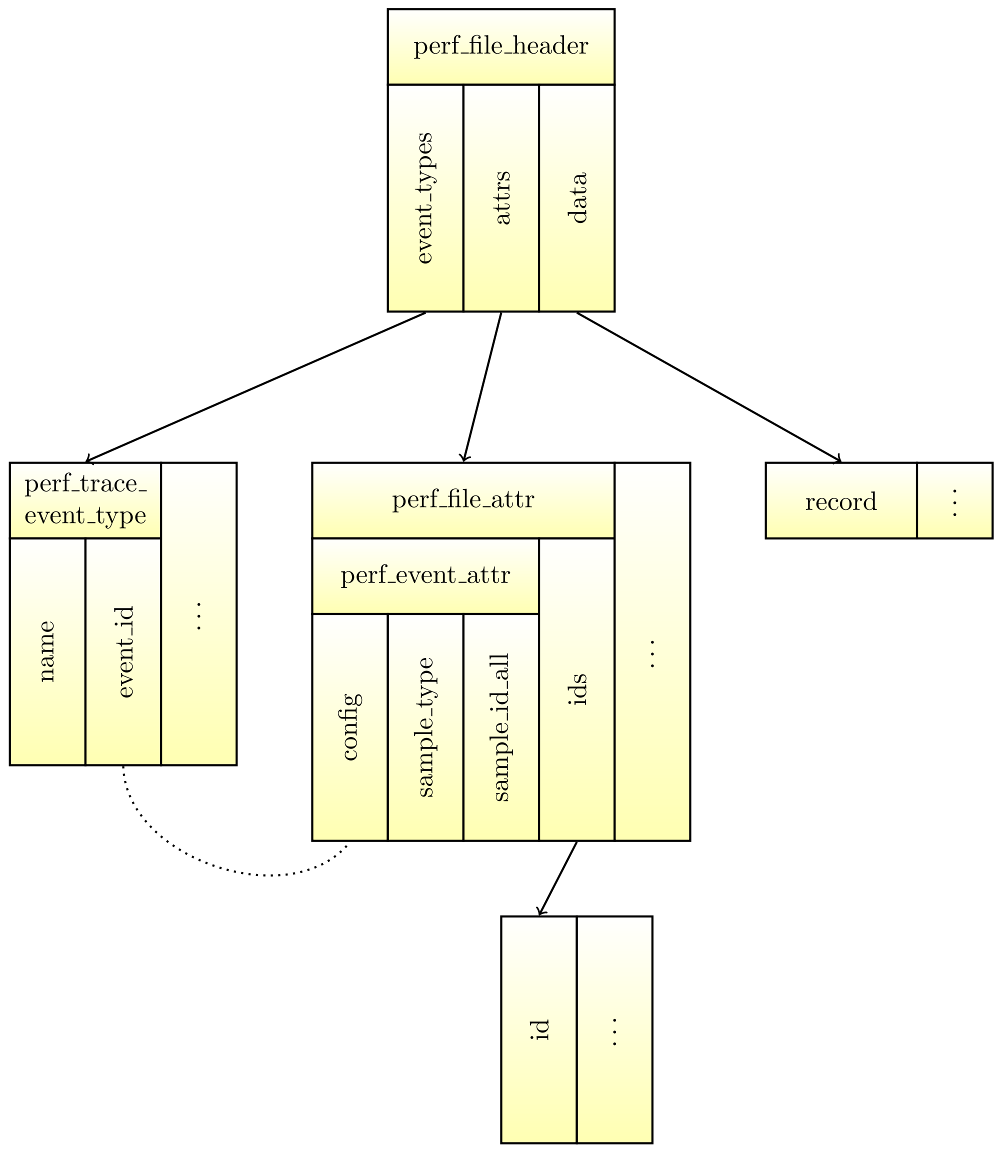

Figure 3: Perf file header. Not all fields of the structures are shown. Links through file offsets are drawn as arrows. Dots in the fields means that the structure can occur more than once. The number can be calculated with the size field and the structure size. Dotted lines means a logical connection between elements.

type	name	description
u32	type	"Major type: hardware/software/tracepoint/etc."
u32	size	size of this structure
u64	config	Link to `.event_id` of `perf_trace_event_type`. See table 3.
u64	sample_period	number of events when a sample is generated if `.freq` is not set
	sample_freq	frequency for sampling if `.freq` is set
u64	sample_type	gives information about what is stored in the sampling record (table 10)
u64	read_format	
u1	disabled	"off by default"
u1	inherit	"children inherit it"
u1	pinned	"must always be on PMU"
u1	exclusive	"only group on PMU"
u1	exclude_user	"don't count user"
u1	exclude_kernel	"ditto kernel"
u1	exclude_hv	"ditto hypervisor"
u1	exclude_idle	"don't count when idle"
u1	mmap	"MMAP" records are included in the file
u1	comm	"COMM" records are included in the file
u1	freq	if set `sample_freq` is valid otherwise `sample_period`
u1	inherit_stat	"per task counts"
u1	enable_on_exec	"next exec enables"
u1	task	"trace fork/exit"
u1	watermark	"wakeup_watermark"
u2	precise_ip	"0 - SAMPLE_IP can have arbitrary skid" "1 - SAMPLE_IP must have constant skid" "2 - SAMPLE_IP can have arbitrary skid" "3 - SAMPLE_IP must have 0 skid" "See also PERF_RECORD_MISC_EXACT_IP"
u1	mmap_data	"non-exec mmap data"
u1	sample_id_all	If set, the records as described in section 3.2 have additional information. We assume the bit is set.
u45	__reserved_1	
u32	wakeup_events	"wakeup every n events"
	wakeup_watermark	"bytes before wakeup"
u32	bp_type	
u64	bp_addr	
	config1	"extension of config"
u64	bp_len	
	config2	"extension of config1"

Table 5: `perf_event_attr` from <*system include directory*>*/linux/perf_event.h*. The quoted text for descriptions is taken from the source code.

3.2 Data

The data section consists of a stream of records, figure 4 gives an overview of the involved data structures.

The data section of the sampling file contains the stream of records coming from the `perf_events` interface (see also [2]). This happens in the function `mmap_read` of the file *util/evlist.c*. Every record has the header as described in table 6. With the size attribute in this structure, one knows the position of the next record.

type	name	description

u32	type	value from enumerator `perf_event_type`:
		`PERF_RECORD_MMAP`
		`PERF_RECORD_COMM`
		`PERF_RECORD_EXIT`
		`PERF_RECORD_FORK`
		`PERF_RECORD_SAMPLE`

u8	misc:0-7	one of the values:
		`PERF_RECORD_MISC_CPUMODE_MASK`
		`PERF_RECORD_MISC_CPUMODE_UNKNOWN`
		`PERF_RECORD_MISC_KERNEL`
		`PERF_RECORD_MISC_USER`
		`PERF_RECORD_MISC_HYPERVISOR`
		`PERF_RECORD_MISC_GUEST_KERNEL`
		`PERF_RECORD_MISC_GUEST_USER`

u6	misc:8-13	unused
u1	misc:14	`PERF_RECORD_MISC_EXACT_IP`, "Indicates that the content of PERF_SAMPLE_IP points to the actual instruction that triggered the event."
u1	misc:15	`PERF_RECORD_MISC_EXT_RESERVED`, "Reserve the last bit to indicate some extended misc field"
u16	size	size of this record (inclusive header)

Table 6: `perf_event_header` from *<system include directory>/linux/perf_event.h*.

For `PERF_RECORD_COMM` in `.type` of the record header, the structure `comm_event` as in table 7 is used. It contains the application name of a process. There should be one or zero comm records for one execution of an application.

type	name	description
u32	pid	process id
u32	tid	thread id
char[16]	comm	name of the application

Table 7: `comm_event` from *<perf source>/util/event.h*.

For `PERF_RECORD_MMAP` in `.type` of the record header, the structure `mmap_event` as in table 8 is used. It contains a used binary (application or library) of a process. With the `.start` and `.len` field one knows the memory location of the binary referenced in the field `.filename`. Together with the instruction pointer from the sample record (table 10) the sample can be assigned to a binary.

type	name	description
u32	pid	process id
u32	tid	thread id
u64	start	start of memory range
u64	len	size of memory range
u64	pgoff	probably page offset, it is used to relocate the memory range
char[PATH_MAX]	filename	binary file using this range

Table 8: `mmap_event` from *<perf source>/util/event.h*.

For `PERF_RECORD_FORK` or `PERF_RECORD_EXIT` in `.type` of the record header, the structure `fork_event` as in table 9 is used. A fork record shows that a new process or thread is created, a exit record shows that a process or thread was terminated.

type	name	description
u32	pid	process id
u32	ppid	parent process id
u32	tid	thread id
u32	ptid	parent thread id
u64	time	timestamp

Table 9: `fork_event` from *<perf source>/util/event.h*.

For `PERF_RECORD_SAMPLE` in `.type` of the record header, the structure `perf_sample` as in table 10 is used. As it can be seen in the table, not all fields of the structure are stored in the file. The function `perf_event__parse_sample` from *<perf source>/util/evsel.c* is used to decode the structure from the file stream. The type is taken from `perf_event_attr .sample_type`. One can see that we need the type to decode the structure to get the id which is used to assign the sample to an `perf_event_attr` entry. But we don't have the type a priori because we don't know to which `perf_event_attr` entry the sample belongs. To overcome this problem, we assume that all `perf_event_attr` entries have the same value for `.sample_type`.

The sample record contains information about event counters. In the `.period` field, the number of events during the sampling time is stored. With the instruction pointer and process id the sample can be assigned to an binary file.

The `id_sample` is not a real structure. It is used to add information to the mmap, comm and fork records. Since it is a subset of `perf_sample`, the same structure is

type	name	valid if flag in `.sample_type`	description
u64	ip	PERF_SAMPLE_IP	instruction pointer
u32	pid	PERF_SAMPLE_TID	process id
u32	tid		thread id
u64	time	PERF_SAMPLE_TIME	timestamp
u64	addr	PERF_SAMPLE_ADDR	
u64	id	PERF_SAMPLE_ID	identification
u64	stream_id	PERF_SAMPLE_STREAM_ID	
u32	cpu	PERF_SAMPLE_CPU	used CPU
u32	res		
u64	period	PERF_SAMPLE_PERIOD	nr. of events
read_format	values	PERF_SAMPLE_READ	
u64	nr	PERF_SAMPLE_CALLCHAIN	
u64	ips[nr]		
u32	size	PERF_SAMPLE_RAW	
char	data[size]		

Table 10: `perf_sample` from <*perf source*>*/util/event.h*. If a flag is set, then the fields are in the file stream. If not, one has to proceed with the next field.

used. The valid fields are shown in table 11. The decoding is done by the function `perf_event__parse_id_sample` from <*perf source*>*/util/evsel.c*. The function is automatically called for the function `perf_event__parse_sample` when the record is not from the type `PERF_RECORD_SAMPLE`.

It is not entirely clear what the `.timestamp` field in an sample contains. Experiments have shown that it may be the running time in nanoseconds of the computer (not uptime as the counter did not run during hibernation). Information suggest that the timestamp is calculated with the Kernel function `sched_clock()`. Nevertheless the source of the timestamp is not clear, it was measured as a strictly increasing series of numbers which is used in perf to sort the records.

type	name	valid if flag in `.sample_type`	description
u32	pid	PERF_SAMPLE_TID	process id
u32	tid		thread id
u64	time	PERF_SAMPLE_TIME	timestamp
u64	addr		
u64	id	PERF_SAMPLE_ID	identification
u64	stream_id	PERF_SAMPLE_STREAM_ID	
u32	cpu	PERF_SAMPLE_CPU	used CPU
u32	res		

Table 11: id_sample

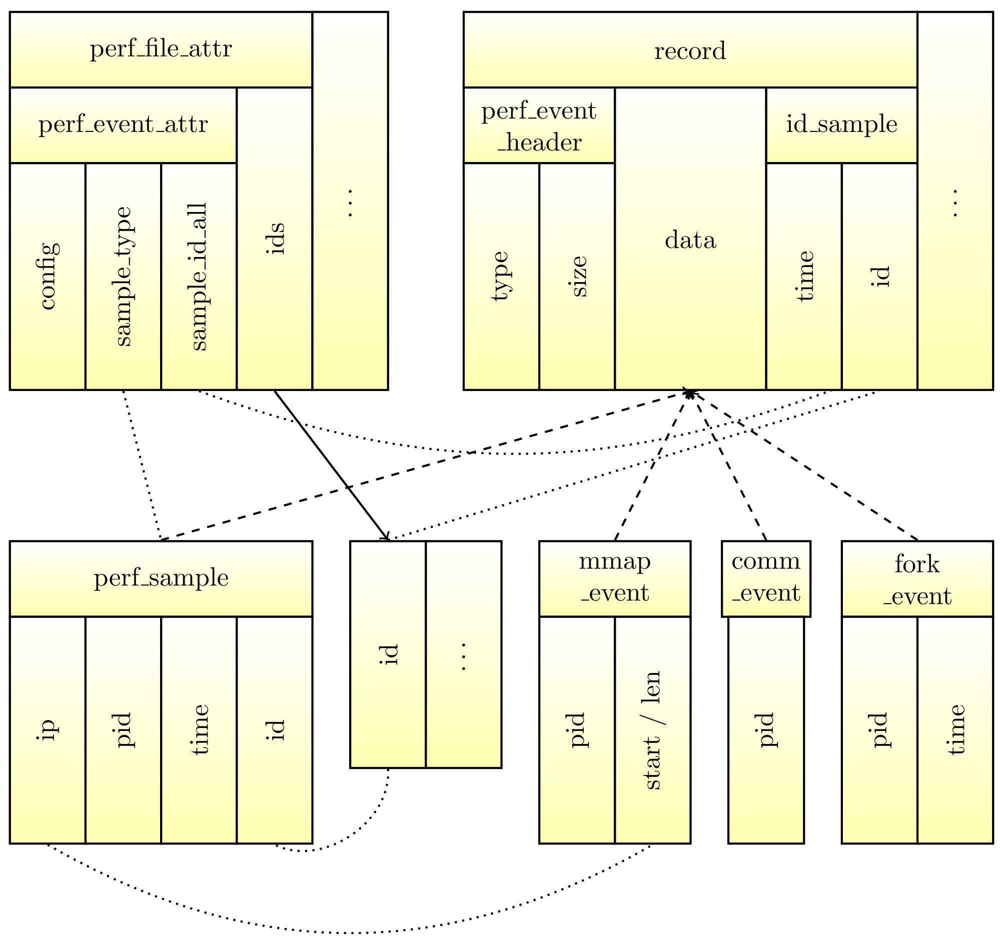

Figure 4: Perf file data. Not all fields of the structures are shown. Links through file offsets are drawn as straight arrows. Dotted lines mean a logical connection between elements. The logical connection between the pid fields and also between the time fields are not shown. The dashed lines mean, that for every record the data is one of the depicted structures.

4 Reading perf files

In this section, a description is given of how the perf data file can be read. For this, an application named readperf is presented. The goal of readperf is not to be used as a tool to analyze the data file, as perf report can be used for this. It is meant to show how the data file can be processed. In addition, it is proof that the data format is understood.

4.1 Using readperf

The command line application to read the perf file is called readperf. It takes exactly one argument, the file name of the perf data file. If no error occurs, an overview of the functions and the percentage of the period is written to the console. After processing the data file, four comma separated files, as described in the following list, are produced.

stat.csv Lists how many records of the different types were found.

overview.csv Content of the data file as a table, sorted by the timestamp. The “nr” column contains the index of the record in the perf data file. The content of “type”, “pid”, “tid” and “time” is clear from the name. Depending of the type, info has a different meaning. For “MMAP”, it contains the filename, address, size and offset (see table 8). “COMM” has the application name as info (see table 7). “FORK” contains the parent pid (see table 9) and “EXIT” has no information. Finally “SAMPLE” has the instruction pointer and period of the sample (see table 10).

processes.csv Every line contains a process. It provides the name of the process, the number of “MMAP” entries, the fork and exit time, the number of samples and the accumulated period.

results.csv This is the file with the most processed data. It contains the accumulated period and number of samples for all used functions as also the source file name of this function.

4.2 Source code

It is written in C and has a Makefile for compiling it. In addition, there are some Doxygen comments in the files. It consists of several source files, the responsibilities is described in the following list:

readperf.c main file, handling of input and output, starting the process

util/tree.h implementation of an AVL tree, used for several structures

util/types.h definition of several used data types

util/errhandler.c routines and data types for error handling

util/origperf.c definition of data types and functions from the original perf source

perffile/session.c initializing and reading of content of the perf file

perffile/overviewPrinter.c functions to log records to an file

perffile/records.c data types and functions to store and iterate the records sorted by the timestamp

perffile/perffile.c reads the content of the file and adds the records to its internal data structure

decode/processes.c handles a data structure of processes sorted by pid, also contains related information like memory maps

decode/processPrinter.c functions to print content of *perffile/processes.c*

decode/addr2line.c function to translate an address of an binary file to the corresponding source file name and source function name

decode/funcstat.c stores source file name and function as well as the corresponding number of samples and period assigned to this function

decode/buildstat.c iterate through the record data structure and build process data structure, update period and sample count of source functions

4.3 Workflow

An broad overview of the workflow can be found in figure 5. The following descriptions are executed in chronological order. It is a short description of the readperf source code.

4.3.1 start_session (session.c)

First of all, the perf file header (table 1) has to be read. This is done with the function `start_session` of the file *session.c*. Testing `.magic` for the content "PERFFILE" ensures that we are really reading a perf file. Comparing the `.attr_size` with the size of the structure `perf_file_attr` gives information whether the values have to be swapped. For readperf, we assume this is not the case.

4.3.2 readAttr (session.c)

To read the attributes into memory we first have to get the number of attribute instances of the structure `perf_file_attr` (table 4). To achieve this, `.attrs.size` is divided by the size of the containing structure `perf_file_attr`. Then we can read the array of instances from the file offset `.attrs.offset`. For every instance we have to read the corresponding IDs. As for the whole structure, there can be several ID's. `.ids.size` is used to determine the number of IDs. If only one event source was used, there is no ID entry since all records belong to the single one `perf_file_attr` instance.

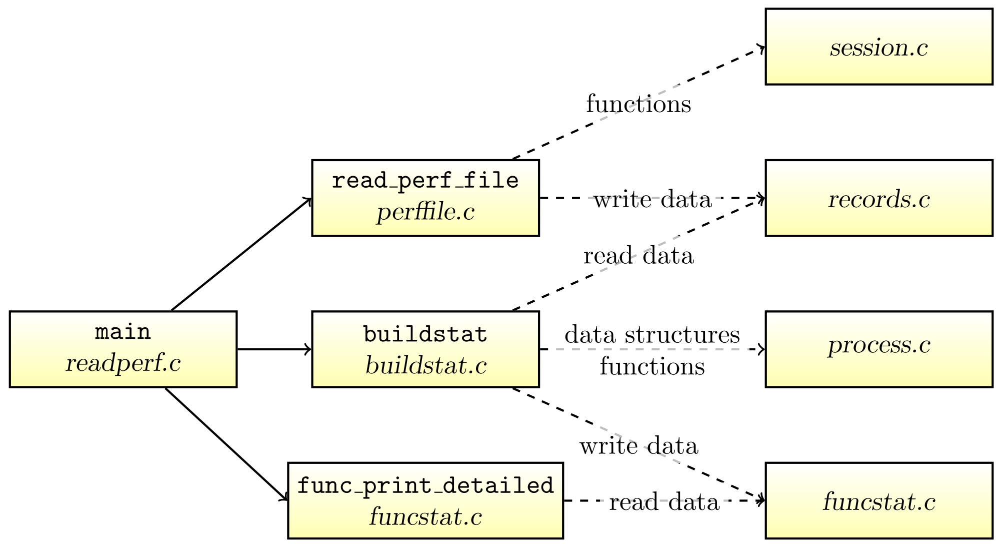

Figure 5: Workflow of readperf. `main` calls the functions `read_perf_file`, `buildstat` and `func_print_detailed`. Those functions use data structures and functionality of further files, depict as dashed lines.

We check that `.attr.sample_id_all` is set for all instances. This ensures that all records have an timestamp and an identification entry. All instances are checked that they have the same value for `.attr.sample_type`.

4.3.3 readTypes (session.c)

There can also be several instances of the `perf_trace_event_type` (table 3) in the file. As before, the `.event_types.size` is used to determine the number of instances. By comparing `.config` of the `perf_file_attr` instances with `.event_id` of the `perf_trace_event_type` instances the corresponding pairs are searched. `.name` from the latter is assigned to the `perf_file_attr` instance.

4.3.4 readEvents (perffile.c)

After the file header is read, the records can be read. We iterate through all records in the file. The ID, timestamp and more are decoded for every record by the function `perf_event__parse_sample`. Specific information for the different types of the record are also decoded and written to a new record. This new record is then stored, sorted by the timestamp.

4.3.5 buildstat (buildstat.c)

Since all records are now sorted in the memory, we can process them. For every record, the corresponding callback function is called. Two new data structures are kept in memory: one to keep track of the actual processes together with memory maps of it and used libraries, and the other to gather the period and sample number for each source function.

4.3.6 decodeFork (buildstat.c)

A new process or thread is created. We check if we already have a process with this pid stored. If yes and the fork created a new process we throw an error because we cannot have two running processes with the same pid. If no process is found and the fork created a thread we also throw an error, since a thread cannot be created without a corresponding process. If a new process is created by the fork, we also create a new process in memory and assign the corresponding pid and timestamp.

4.3.7 decodeExit (buildstat.c)

A process or thread is terminated. If it was a process, it is removed from the internal list of processed and the information is written to a file.

4.3.8 decodeComm (buildstat.c)

Provides the application name for an process. If the corresponding process is not found we assume that it was not yet created. This is the case for processes running at the time perf record was started. If so, we expect the timestamp to be zero and create the process. The name, provided by the record, is assigned to the process.

4.3.9 decodeMmap (buildstat.c)

A library module was loaded. As for "COMM" records, it is possible that a process does not yet exist. For that case we create one as in the function `decodeComm`. The information of the record is added to the process. If the `.filename` is `[vdso]` we assume that this record contains the begin of the address space of the libraries. In this case, the `.pgoff` information is stored as `.vdso` for the process.

4.3.10 decodeSample (buildstat.c)

A new sample has been produced. The corresponding process is searched for, if not found, we assume it belongs to a common process with the pid ffffffff. The number of samples of this process is increased by one and the period of the record is added to the period of the process.

In addition, the application or library where the `.ip` of the sample points to is searched within the mmap entries of the process. If it is a library we subtract the start address of the library from the instruction pointer to get the address. For an application, we just use the instruction pointer. This address together with the binary name is used to search for or create the source function name where this event occurred. As for the process, the sample count and period of the function is updated.

4.3.11 force_entry (funcstat.c)

Returns an entry which identifies a source function together with the source file and additional information like the sample count and period. First, it searches for an entry with this binary name and instruction pointer. If not found, it retrieves the source file name and source function name and searches for an entry with that. If this also does not leads to an valid entry, a new one is created.

4.3.12 get_func (addr2line.c)

Returns an source file name and function name to an instruction pointer / binary name pair. At the moment, it uses the GNU Binutils tool addr2line.

4.3.13 func_print_detailed (funcstat.c)

This function prints a list of function names together with the source file name, sample count and period.

5 Conclusion

For Linux, perf is the default way to measure performance. Although a tool for reporting is provided, it may not cover all possible use cases. For this reason, one has to understand how the system works.

In this report, an overview of performance monitoring and the Linux tool perf was given. The data file produced by this tool was inspected. All required data structures were analyzed and described.

A tool called readperf was written to show how one can read the data file. It produces several output files. All of them are comma separated tables. One of them is a complete list of all records, sorted by the timestamp. The tool can also resolve the instruction pointer of the samples and through that assign the samples to a source code function. This is then the final, most processed output of readperf.

6 Further work

The execution speed of readperf compared with perf report is quite slow. This mainly comes from the fact that readperf starts the external tool addr2line to translate an instruction pointer to the source file function name. Since perf report is much faster, there exists a better solution to do that.

As mentioned before, readperf can only handle one event source. It should be an easy task add support for multiple events. To do that, the event source has to be found with the function `get_entry` of the file *<readperf source>/perffile/session.c*. This can be done in the function `readEvents` of the file *<readperf source>/perffile/perffile.c* or `handleRecord` in *<readperf source>/decode/buildstat.c*. The file writing functions have to be changed too.

At the moment, the whole data file is loaded into memory and the processed. This is not the best solution for two reasons. Firstly, a data file can be quite big. Second, a tool would maybe process data online, just during capturing (and not storing the whole file). The problem is that the records are not sorted by timestamp. But it seems that there exists a way to know when it is safe to process a bunch of records. To do that, one has to know which timestamp is a lower bound for all future timestamps. Figure 6 supports the idea of a lower bound timestamp. The function `perf_session_queue_event` in the file *<perf source>/util/session.c* may be a starting point.

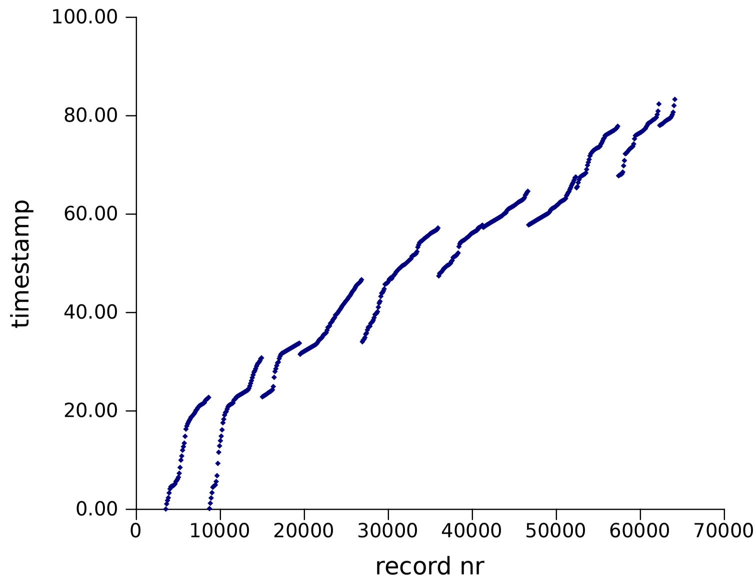

Figure 6: Timestamp depending on the entry in the data file. It was recorded on a two core system. Only every 100th entry is shown. Timestamp is divided by 10^9 and the start offset is subtracted. It can be seen that there exists a clear lower and upper bound for timestamps.

If one is only interested in processing the data from the data file, the callback functions can be used. After installed the callback functions, they are called with an occurrence of an record in the data file. As an example, *<perf source>/builtin-report.c* can be used.

List of Figures

List of Tables

References

[1] Reza Azimi. *Hardware Performance Monitoring*. 2009. URL: www.cse.shirazu.ac.ir/~azimi/perf88/lectures/Lect5-HardwarePerfMon.pdf.

[2] Stephane Eranian. *perf_events status update*. Aug. 2010. URL: http://cscads.rice.edu/workshops/summer-2010/slides/performance-tools/perf_events_status_update.pdf/view.

[3] Sverre Jarp. *Computer Architecture and Performance Tuning*. Sept. 2010. URL: http://indico.cern.ch/getFile.py/access?resId=1&materialId=slides&confId=36801.

[4] kernel.org. *Linux kernel profiling with perf*. June 2011. URL: https://perf.wiki.kernel.org/index.php/Tutorial.

[5] Arnaldo Carvalho de Melo. *The New Linux 'perf' Tools*. Tech. rep. 2010. URL: http://vger.kernel.org/~acme/perf/lk2010-perf-paper.pdf.

[6] Andrzej Nowak. *CERN openlab Computer Architecture and Performance Tuning Workshop*. 2011.

[7] Vince Weaver. *The Unofficial Linux Perf Events Web-Page*. 2011. URL: http://web.eecs.utk.edu/~vweaver1/projects/perf-events/.