A Study of Linux Perf and Slab Allocation Sub-Systems

by

Amir Reza Ghods

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Today, modern processors are equipped with a special unit named Performance Monitoring Unit (PMU) that enables software developers to gain access to micro-architectural level information such as CPU cycles count and executed instructions count. The PMU provides a set of programmable registers called hardware performance counters that can be programmed to count the specific hardware events. In the Linux operating system, many low-level interfaces are designed to provide access to the hardware counters facilities. One of these interfaces is *perf_event*, which was merged as a sub-system to the kernel mainline in 2009, and became a widely used interface for hardware counters.

Firstly, we investigate the perf_event Linux sub-system in the kernel-level by exploring the kernel source code to identify the potential sources of overhead and counting error. We also study the Perf tool as one of the end-user interfaces that was built on top of the *perf_event* sub-system to provide an easy-to-use measurement and profiling tool in the Linux operating system. Moreover, we conduct some experiments on a variety of processors to analyze the overhead, determinism, and accuracy of the Perf tool and the underlying perf_event sub-system in counting hardware events. Although our results show 47% error in counting the number of taken branches as well as 5.92% relative overhead on the Intel Pentium 4 processors, we do not observe a significant overhead or defect on the modern x86 and ARM processors.

Secondly, we explore a memory management sub-system of Linux kernel called slab allocator, that plays a crucial role in the overall performance of the system. We study three different implementations of the slab allocator that are currently available in the Linux kernel mainline and enumerate the advantages and disadvantages of each implementation. We also investigate the binning effect of the slab allocator on the Linux system calls execution time variation. Moreover, we introduce a new metric called "Slab Metric" that is assigned to each system call to represent the interaction level with the slab allocator. The results show a correlation coefficient of 0.78 between the dynamic slab metric and the execution time variation of the Linux system calls.

Acknowledgements

This thesis would not have been possible without the support of many people. I would like to thank my supervisor, Prof. Fischmeister for his advice and support throughout my thesis work. I would also like to thank Jean-Christophe Petkovich for his openness to my queries. Finally, I would like to extend my sincerest thanks to James Millar from CMC Canada for his assistance in purchasing the trace hardware equipment.

Dedication

I want to dedicate this thesis to my parents for their endless love and support.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

PMU Performance Monitoring Unit iii
NUMA Non-Uniform Memory Access 1
SMP Symmetric Multi Processing 1
MSR Model Specific Register 3
TLB Table Lookaside Buffer 3
HPC Hardware Performance Counters 14
DBI Dynamic Binary Instrumentation 23
ISA Instruction Set Architecture 22
PTM Program Trace Macrocell 24
ETM Embedded Trace Macrocell 24
ITM Instrumentation Trace Macrocell 24
PFT Program Flow Trace 25
ASID Address Space Identifier 27
MAD Mean Absolute Deviation 29
RSD Relative Standard Deviation 58
GFP Get Free Page 43
CV Coefficient of Variation 58
TSC Time Stamp Counter 6

Chapter 1

Linux Perf: Linux Performance Evaluation Tool

1.1 Introduction

Over the past several decades, computer hardware designers have attempted to keep pace with the high demand for processing power by developing a new generation of processors, which involves improving computer architecture, design, and implementation. Also, enhancements in other hardware components, such as main memory and internal bus, that work closely with the CPU can increase the overall performance of the system. However, we cannot always rely on hardware to deliver increased performance. At some point, we need to create high-performance programs or improve our current programs to make the most of the available hardware resources. Moreover, with the recent changes to the processors and memory architecture such as Symmetric Multi Processing (SMP) and Non-Uniform Memory Access (NUMA), developers may experience performance issues after deploying their programs to the new generation of hardware. For instance, software programs that use algorithms that are not optimized for running on the SMP environment may face serious performance degradation as a result of lower CPU frequency and the overhead imposed by CPU cores' synchronization in the operating system level. Therefore, developers need to evaluate their software carefully to spot performance issues by conducting measurement-based analysis and find the root cause of the problem through the performance debugging tools.

Software performance analysis is one of the topics in computer science that is always important to both individuals and industries. Performance-related bugs are the toughest

to detect in software when a small change in the source code can cause a huge performance degradation in the software. Rigorous measurement-based software performance analysis is a method that can reveal software regression in the early stages of its development. In rigorous measurement-based performance analysis, before releasing a new version of the program, we run the program under a particular workload and measure the performance-related metrics to make sure that the latest changes to the program did not impose any unpredicted performance degradation. Many tools and methods are developed to help the developers to find performance bottlenecks in the software. In the Linux operating system, we have a variety of tools such as Perf, ftrace, strace, top and tcpdump. that are designed for monitoring a specific part of the system. However, in this work, we are only interested in identifying the performance issues that are related to the execution of the programs on the CPU under a particular workload. These types of improvement and performance tuning can be achieved by performing measurement-based performance analysis.

In general, three commonly used approaches are available for performing measurement-based performance analysis:

- Sample based: The *Sample based* method is a low price method that offers a relatively accurate results with low overhead. This method is suitable for root cause performance analysis. Although all sampling-based techniques are associated with blind spots, we still can use this method in most of the performance analysis projects since the we are only interested in finding the time-consuming functions (hotspots).

- Instruction level Instrumentation: *Instrumentation* method provides a highly accurate and low price technique in spite of high overhead that makes it impractical in a real-time context.

- Hardware level trace: This method offers a 100% non-intrusive trace capture at the finest possible granularity and accuracy. However, both technical and practical limitations of this method in conjunction with the high price of the tracing equipment, makes this approach less viable for conducting regular performance analysis. Also, since we capture traces directly from the hardware, this method requires us to post process the traces the traces to make them readable and meaningful to the end-user.

Table 1.1 shows a basic comparison between these three methodologies.

Choosing an appropriate method for conducting measurement-based performance analysis requires a special care since each approach has its advantages and disadvantages.

Method	Cost	Overhead	Acuuracy	Granularity	Easy to Use
Hardware Trace	High	Low	High	High	Low
Instrumentation	Low	High	High	Medium	High
Sample-based	Low	Midium	Low	Low	High

Table 1.1: Comparing performance analysis methods

Hardware performance counters made a significant contribution to the state-of-the-art in the field of performance evaluation. It becomes a dominated measurement based approach for performance evaluation of the programs that provides micro-architectural level information including CPU cycles, CPU stall cycles and Table Lookaside Buffer (TLB) misses, directly from the CPU. Hardware performance counters (also known as Model Specific Register (MSR) in x86-64 architecture) are a set of registers integrated into the modern microprocessors that enable one to count the number of hardware events that occurred during an execution of a program. We usually have two types of hardware performance counter registers namely, "counter" registers that keep the number of occurred events and "control" registers that are being used for selecting the hardware events, overflow interrupts and controlling the counter registers.

We can distinguish the contribution of the Linux community around making use of hardware performance counters into the following three categories:

- First, drivers development in different architectures that enables access to the Performance Monitoring Unit (PMU) through the machine dependent instructions.
- Second, developing a kernel-level interface that uses PMU drivers to provide a safe access to the hardware counters via the system calls. These type of developments are usually published as a patch for the Linux kernel. Usually, a kernel re-compilation is needed for using these kernel interfaces.
- Third, the user-space level interfaces that are created on top of the kernel interfaces that provide a high-level and easy-to-use tools for end-users. These programs enable end-users to analysis their applications based on events count measurement using hardware performance counters. Other core features in these programs include: performance profiling, events count, call graph and advance performance reporting.

The following diagram(1.1) illustrates the hierarchy of access levels to the hardware performance counters on the Linux operating system.

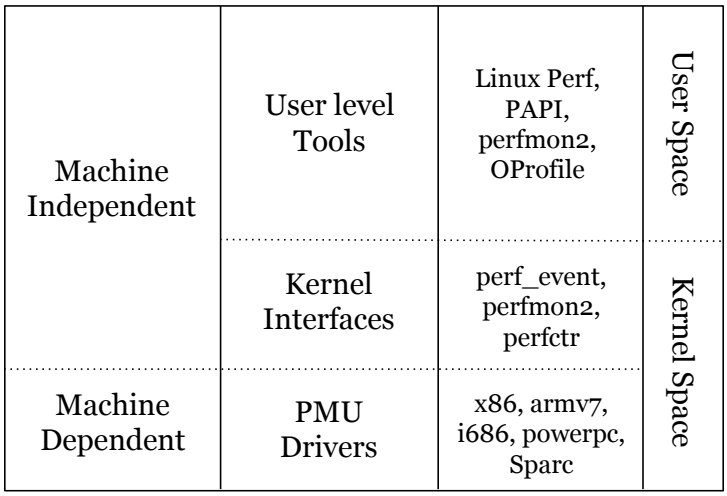

Figure 1.1: Hardware Performance Counter Hierarchy

1.2 Problem Statement & Approach

In this work, we study the Linux Perf tool as an end-user interface that built on top of the `perf_event` Linux sub-system. To be more specific, we investigate the accuracy, determinism and the overhead of the Linux Perf in utilizing the hardware counters as an underlying mechanism for conducting the measurement-based performance analysis. Our approach to evaluating the accuracy of the Perf is to compare the results of Perf with the results we obtained from other approaches such as *Hardware Trace* and *Instrumentation* that will be explained in detail in Section 1.6.

1.3 Literature Review

The very first appearance of hardware performance counters in literature dates back to 1994, specifically, Terje Mathisen refers to `RDMSR` and `WRMSR` assembly instructions as the secrets of Intel Pentium series processors [6, 35].

In order to make use of hardware performance counters in the Linux operating system, patches appeared soon after the release of Intel Pentium processors enabled the Linux kernel to access the CPU PMUs [36, 25, 18, 29, 28, 27, 22, 30]. Although these early

implementations allowed Linux users to access the hardware counter registers, a few of them became popular among the Linux community.

The first well-stablished interface for accessing hardware performance counter facilities was PAPI, which was introduced in 1999 [38]. In the early versions of PAPI, designers implemented both a low-level hardware counters interface (as a kernel extension) and a high-level interface for novice users. However, in recent versions of the tool, they switched to other well-stablished low-level drivers for access to the hardware performance counters. Currently, PAPI is using Permon2, Perfctr for the Linux kernel version 2.6.30 (and below), and the perf_event Linux kernel official sub-system for working with hardware performance counters.

OProfile [33] is the name of another Linux-based tool written in C++ that provides a high-level interface for end-users. It is also used to provide a low-level interface to the hardware performance counters through patching the Linux kernel, but it recently adopted perf_event sub-system as the low-level interface for accessing performance counters. In the latest version of OProfile, it supports both events sampling and aggregation mode, stack trace analysis, per-process and system-wide profiling.

The following two interfaces were added to the Linux kernel as a patch to provide a low-level access to the PMU in a variety of processors.

1. Perfctr: *Perfctr* [37] is a widespread low-level interface for accessing to the hardware performance counters in Linux 3.6.x that was introduced in 1999. It provides access to the performance counters through a device node in `/dev/perfctr`. *Perfctr* is suitable for self-monitoring and basic sampling support and provides per-thread and system-wide monitoring. An advantage that Perfctr has over Perfmon2 and perf_event is its ability to read the value of performance counters using `readpmc` instruction which is much faster than invoking a system call. This interface was used by PAPI before introducing Perfmon2 and perf_event.

2. Perfmon2: *Perfmon2* [24] is flexible performance monitoring interface for Linux that provides a generic interface to access the processors' PMU. *Perfmon2* uses a helper library called *libpfm* that works in kernel level and provides an abstracted model to access performance counters on a broad range of hardware. In the earlier versions of Perfmon2, it contained twelve system calls that were reduced to four system calls a after code review conducted by the Linux community. Caliper(HP), PAPI and pfmon end-user performance analysis tools use Perfmon2 interface underneath. Eventually, in 2009, Perfmon2 was abandoned in favor of perf_event sub-system in Linux kernel.

In 2009, the *perf_event_open* system call was added to the mainline Linux souce code (version 2.6.32) as a kernel component. In 2013, Vincent M. Weaver elaborated on perf_event features and overhead in comparison with other hardware performance counter interfaces. In counting context switch event, he reported up to 20% overhead in average compared to 2% overhead in perfctr and perfmon2 [42].

Zaparanuks *et al.* [21] investigated the accuracy of PAPI, perfmon2 and perfctr on different machines with the emphasis variation rather than overhead. They found that variations were near similar across different machines (perf_event was not available on that time).

Moreover, Salayandia [40], DeRose *et al.* [23], Moore *et al.* [39] and Maxwell *et al.* [34] studied variability and overhead of hardware performance counters and their underlying operating system interfaces such as PAPI and perfmon2.

Vincent M. Weaver *et al.* evaluated the determinism of CPU PMU on a different implementations of x86_64 architecture and reported an overcount and run-to-run variation on even under highly restricted and controlled environment. He also investigated ARM, SPARC, and POWER PC systems and found the events count more deterministic compared to the x86_64 architecture [41].

In 2015, Vincent M. Weaver explores the overhead of perf_event in self-monitoring with a focus on the time overhead imposed by operating system interface. He uses Time Stamp Counter (TSC) register counter that is available on all x86 processors as a low-overhead measurement method. He has found a significant overhead in an order of magnitude larger than perfmon and perfctr implementations. Also, he proposed a proper coding method to significantly reduce the overhead of perf_event in self-monitoring mode [43].

1.4 Linux perf_event Sub-System

In this section, we briefly explain the `perf_event_open` system call functionality and how it configures the underlying PMU and captures the generated hardware events.

The list below shows the features of perf_event sub-system that are considered in its design process:

- Supporting different counting modes
  - Aggregate or Counting: In this mode, the PMU is configured in such a way that only counts and reports the total number of hardware events during the execution of a program.

  - Sampling: In sampling mode, after counting a specific amount of events, the PMU stops the counter and raises an overflow hardware interrupt to allow the kernel to take a sample record that consists of valuable information of the execution of the program. (i.e., stack frame, program counter)

- Supporting per-thread, per-process and per-CPU monitoring
  - Saves and restores the states of counter registers on each context switch
  - States presists per logical cpus on context switch
  - Monitors events in processor wide mode
  - Capturing Offcore and Uncore events
- Abstracting event-based API
  - Abstracts away PMU registers events name from users
  - Supports software events such as context switches and page faults
- Configuring more events to monitor than the actual available hardware counter registers
  - Supports unlimited number of software events
  - Scales the events count in case of multiplexing
- Event grouping
  - Measures a set of events together in case of having more events to monitor than the actual available counters

1.4.1 `perf_event` Interface

The perf_event sub-system only added *one* system call to the Linux kernel. The *perf_event_open* system call returns a file descriptor to identify the configured event(s). It manages the event(s) independently through file descriptors that allows one to configure and count events with different configurations in a single session [12].

```
int perf_event_open(struct perf_event_attr *attr,
                    pid_t pid,
                    int cpu,
```

```
int group_fd,
unsigned long flags);
```

The first parameter that *perf_event_open* takes as an input has a complex structure with over 45 fields that are being used to describe the events we are interested in counting as well the counting configurations. The second argument is an identifier of the target thread that either specifies the process PID of a particular program or the currently running program for counting in self-monitoring mode. Passing 0 or -1 as the second parameter to the function will enable the self-monitoring mode or the system-wide (count events of the whole system) monitoring mode, respectively. The *cpu* argument limits the measurement to a particular CPU if the specified number is greater than 0. In case of `cpu = 0`, hardware events will be measured on all CPUs. The next argument is the file descriptor of the group leader of the event(s). Events grouping enables one to measure all of the members of the group at the same time. In other words, the measured values of the hardware events that belong to the same group can be meaningfully compared. Providing this option will force the kernel event scheduler to either put all of the group members on the CPU or avoid measuring any group members in case of failure.

1.4.2 `perf_event_open` System Call

In this section, we will take a closer look at the `perf_event_open` system call in the kernel level and will provide the minimum requirements for using the hardware counters in the user level. The `perf_event_open` system call does not have any wrapper in the `libc` library. Therefore, the system call has to be called using the `syscall` function with the `__NR_perf_event_open` as the first parameter.

Most of the machine's independent source code related to `perf_event_open`, including the system call definition, resides in the `"kernel/events/core.c"` file. Also, for each architecture, there are low-level drivers that abstract away access to the hardware counters. In the x86 instruction set, PMU driver uses `rdmsr` and `wrmsrt` instructions to access the MSR [8]. However, in ARM architecture, the PMU registers are accessible through the `CP15` system control coprocessor or external APB interface [5]. Moreover, in the `MIPS32` and `MIPS64` ISAs, the system control coprocessor provides performance counter facilities through the `PerfCtl` and `PerfCnt` control registers.

With over 50 configurations, the `perf_event_open` system call is the heart of the `perf_event` Linux kernel sub-system. It also has the longest manual page among the available Linux system calls. The functionality of this system call is quite different in sampling and counting mode.

In counting mode, the user can control the performance counter with three basic operations: reset, enable and disable.. These operations can be performed by calling the `ioctl` system call with the `perf_event` file descriptor as the first parameter, and the operation number as the second parameter. The Snippet 1.1 shows the minimum required codes for configuring a hardware event in *counting* mode.

```
struct perf_event_attr attrs;
int fd;
long long count;

memset(&pe, 0, sizeof(struct perf_event_attr));
attrs.type = PERF_TYPE_HARDWARE;
attrs.size = sizeof(struct perf_event_attr);
attrs.config = PERF_COUNT_HW_INSTRUCTIONS;
attrs.disabled = 1;        //do not start counting
attrs.exclude_kernel = 1; //exclude kernel count (x86 only)
attrs.exclude_hv = 1;      //do not count hypervisor

pid_t pid = 0;       // measure the current process
int cpu = -1;        // measure on any cpu
int group_fd = -1;  // no event grouping
unsigned long flags = 0;

fd = syscall(__NR_perf_event_open, &attrs, pid, cpu, group_fd, flags);

if (fd == -1) {
  fprintf(stderr, "Failed!");
  exit(-1);
}

ioctl(fd, PERF_EVENT_IOC_RESET, 0); //reset hardware counter
ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);//enable hardware counter

/* start codes to count */
for(long int i = 0; i < 10000000000; i++){
  __asm__("nop");
}
/* end */

ioctl(fd, PERF_EVENT_IOC_DISABLE, 0); //disable hardware counter

read(fd, &count, sizeof(long long)); //reading counter value

printf("count instructions: %lld\n", count);
```

Snippet 1.1: Perf event count in self-monitoring mode

Unlike counting mode, `perf_event` configuration in sampling mode is more complicated in both kernel and user levels. First of all, the `perf_event` sub-system needs to configure the PMU (using `MSRWR` on x86) to overflow when the hardware event count reaches to a specific value called the "sampling period." By invoking the `perf_event_open` system call in sampling mode, the kernel would automatically adjust the sampling period based on the frequency of hardware event generation and the sampling frequency that is provided via the `perf_event_attr.sample_freq` attribute. After each PMU overflow interrupt, the kernel readjusts the sampling period using the formula below as long as we did not specify a fixed sampling period at the time of creating the event.

$$new\_sampling\_period = \frac{last\_sample\_period \times 10^9}{elapsed\_time \times sample\_freq} \tag{1.1}$$

To observe the impact of this adjustment technique on different hardware events in the sampling period, we perform a simple experiment in which we use the *instruction* and the *branch* events that have different event generation rates. Figure 1.2 shows the sampling period of the first 100 samples taken from the hardware events while executing the following command.

```
dd if=/dev/zero of=/dev/zero count=10000000
```

As we expected, the sampling period of the *instruction* hardware events is dramatically increased in comparison to the *branch* event.

After configuring an event for counting in sampling mode, we need to map the file descriptor that returned from the `perf_event_open` to the user address space. The following snippet shows how we can use `mmap` function to map the file descriptor to the program address space that can provide a direct access to the taken samples from the user space.

```
1 char* mmap_address = mmap(NULL, NR_PAGES*PAGE_SIZE, PROT_READ|PROT_WRITE,
    MAP_SHARED, event_fd, 0);
```

`NR_PAGES` is the number of memory pages that is mapped from the kernel ring-buffer. Also, `MAP_SHARED` flags makes this mapping visible to the other processes that enables the programs such as Perf tool to use hardware performance counters for measuring the hardware events of other processes on their behalf.

The next step is to configure the user program that initiated the `perf_even_open` to handle the *wakeup* signal and save the captured samples after the memory mapped pages fill up (or a certain threshold is reached). To be more specific, a wakeup signal will be sent to the user program when either the number of samples that are stored in the ring-buffer

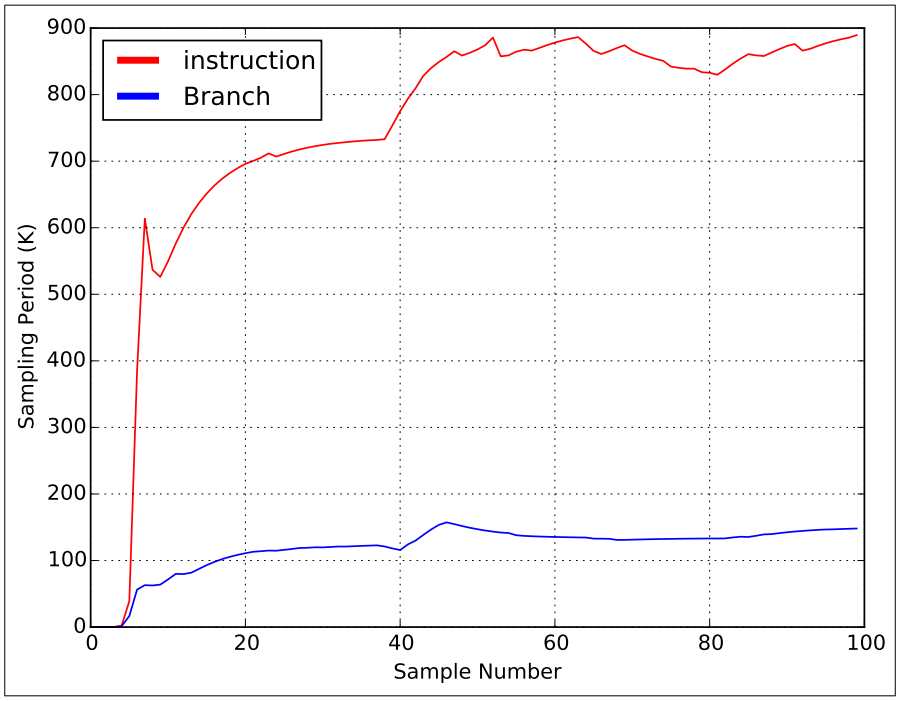

Figure 1.2: Kernel Level Sampling Period Adjustment

reaches the `perf_event_attr.wakeup_events` value, or the ring-buffer consumes all of the memory mapped pages. Wake up conditions can be captured by polling the `perf_event` file descriptor using the `poll` or the `select` system calls over the `POLL_IN` event, or by setting up a signal handler that handles the `SIGIO` signal using the `fcntl` system call. The Linux Perf tool preferred to use the polling method rather than the signal handling as it interrupts the execution flow of the program and makes it exceedingly slow. Once the wakeup event is captured, the user-space program retrieves the pointer that points to the beginning of the buffer (`perf_event_mmap_page->data_head`), and then starts to read the captured samples from the next 2^n pages.

A simplified control flow of a program that configures the PMU to monitor a particular event in sampling mode is depicted in Figure 1.3.

In Figure 1.3, the *wakeup* signal is handled by the wakeup handler function that receives the signal when the buffer becomes ready for reading. There are two potential sources of miscounting which can make the results inaccurate. First, when an overflow occurs in the PMU event counter that disables the counter until the `perf_event` stores the sample into the ring-buffer. For instance, setting the sampling period to a small number for a high-

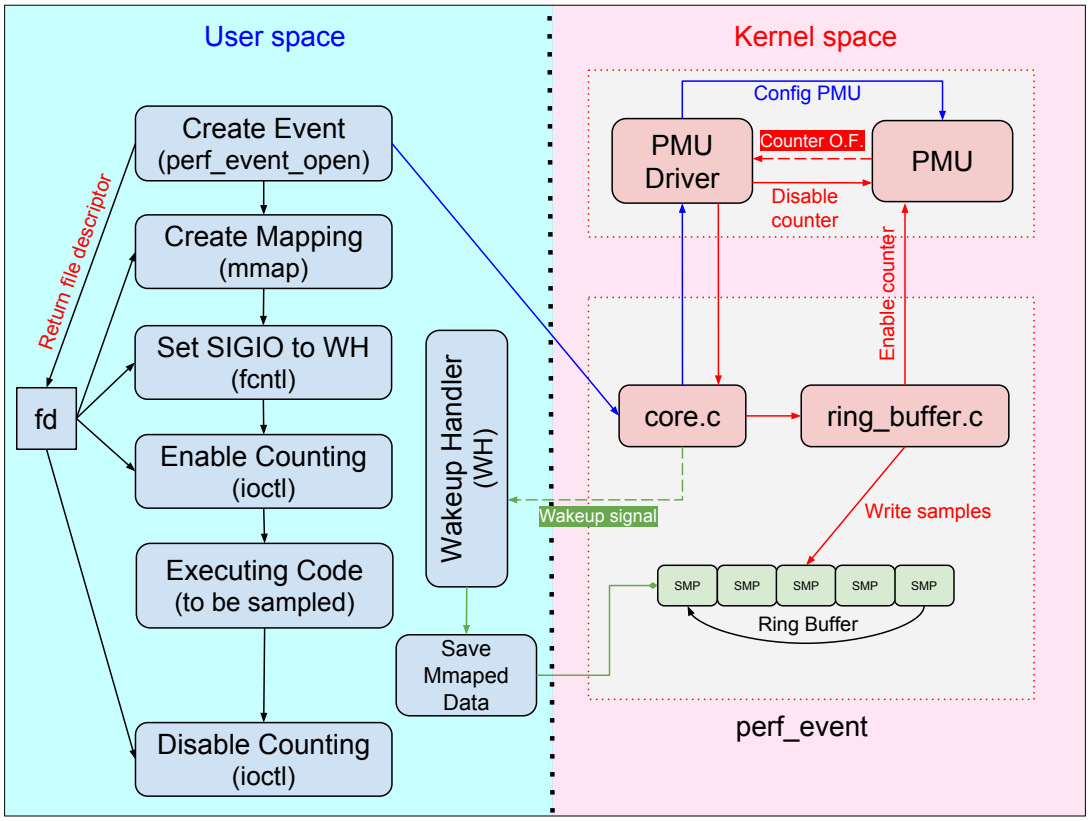

Figure 1.3: Sample-based self-monitoring using perf_event

frequency event such as CPU cycle will generate excess event counter overflow that result in missing a significant number of events. The second root cause of sample miscounting is the *wakeup* signal handler. It can happen when the wakeup handler in the user-space program takes too much time to store the samples. In this case, we will lose samples due to the overwriting of the new samples that coming from hardware into the ring buffer that has the old samples. In Section 1.7.2 we will investigate the impact of the potential sources of miscounting on the accuracy of the Perf in sampling mode.

1.4.3 `perf_event` Supported Events

`perf_event` supports a variety of events from both hardware and software side. The hardware events are coming from the hardware performance counters that are implemented in the chipset. However, the software events are provided by the Linux *uprobe* and *kprobe*

debugging facilities.

Figure 1.4 shows the software and hardware events that are accessible via the `perf_event` sub-system across the Linux operating system.

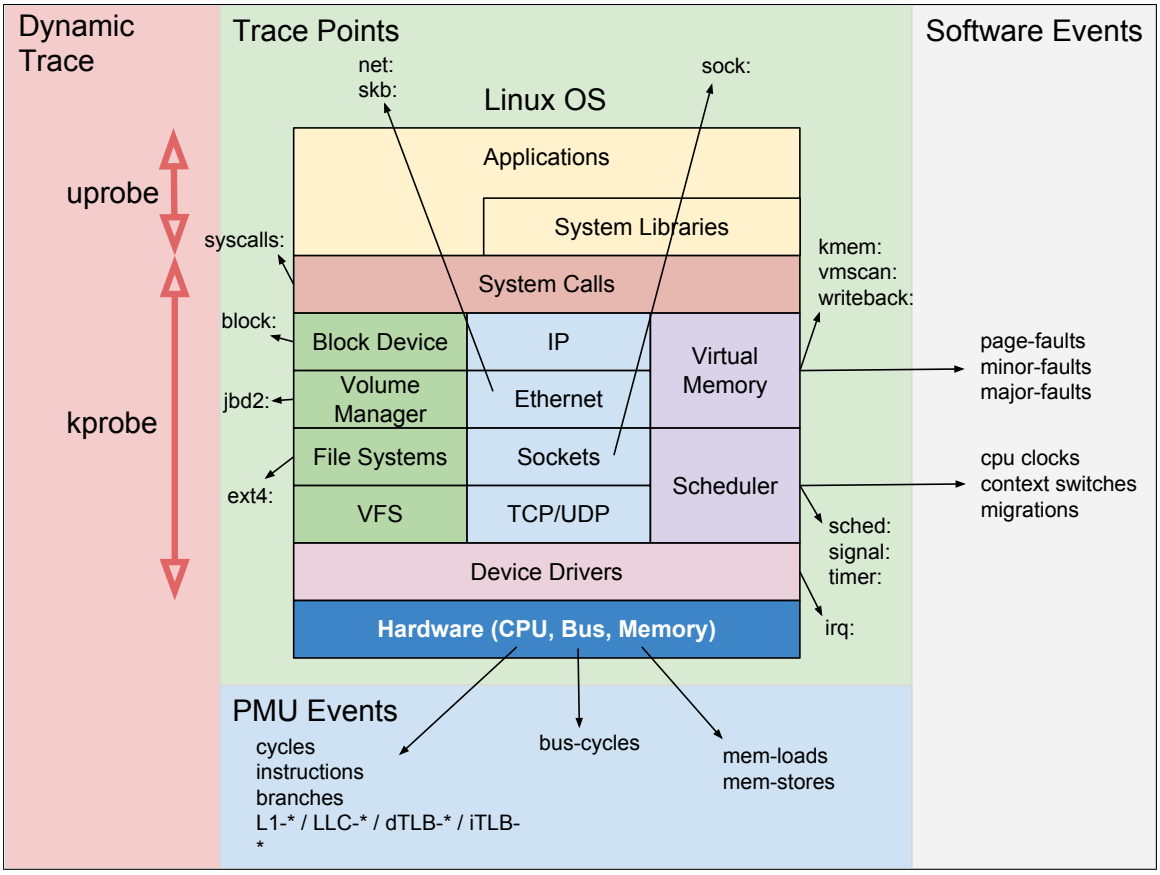

Figure 1.4: perf_event Hardware/Software Events Sources [26]

1.5 Linux Perf Tool

Perf is a Linux-based profiler and performance-measuring tool that was introduced in Linux version 2.6.31. The Perf tool was originally designed to create a tool on top of the `perf_event` sub-system that uses hardware performance counters for counting the hardware events. However, today, it is one of the most important performance evaluation

tools among the Linux community. At the time of writing this thesis, Perf was capable of performing a variety of performance measurements in both kernel space and user-space. It currently supports the following features that are accessible through the following sub-commands:

- Hardware Performance Counters (HPC)
- Software events counters
- Tracepoints
- Advanced reporting
- Analyze lock events
- Dynamic Probes (e.g. uprobes and kprobes)
- Top (system profiling tool)
- Profiling memory accesses
- Strace inspired tool that captures a profile of the invocations to the system calls
- Ftrace that is a front-end for kernel's ftrace
- Annotate for source level analysis

The above list demonstrates that Linux Perf has shifted from only being a front-end for accessing to the hardware performance counters to a powerful collection of the performance evaluation tools that can be used in a variety of hardware to facilitate performance analysis for the Linux performance community.

As previously mentioned,, we will focus on the Linux Perf hardware performance counters features, overhead, and its accuracy in different situations.

We normally use the following three Perf sub-commands to perform the HPC-based measurements in Linux:

- stat: for counting the events → shows the results at the end of the measurement
- record: sample-based events collection → report *perf.data* in binary
- report: parse the *perf.data* file → high level analysis

- anotate: parse *perf.data* file → source level analysis

HPC-based performance measurements can be conducted in two different modes: *Counting* or *Sampling* modes. *Counting* or *Aggrigation* mode is suitable for gathering statistics of a specific process or the entire system in a particular time interval. In this mode, Perf simply aggregates the occurrence of the events and either reports them at the end of the execution of the running process, or when the user sends a `SIGINT` signal to interrupt the Perf's process.

The following shows a sample output of the *perf stat* that runs the *sleep* Linux command:

```
$ perf stat sleep 5
 Performance counter stats for 'sleep 5':

      0.339207      task-clock (msec)         #    0.000 CPUs utilized
             1      context-switches          #    0.003 M/sec
             0      cpu-migrations            #    0.000 K/sec
            59      page-faults               #    0.174 M/sec
     1,048,747      cycles                    #    3.092 GHz
       740,254      stalled-cycles-frontend   #   70.58% frontend cycles idle
       669,591      instructions              #    0.64  insns per cycle
                                              #    1.11  stalled cycles per insn
       139,229      branches                  #  410.454 M/sec
         7,260      branch-misses             #    5.21% of all branches

       5.000645228 seconds time elapsed
```

The results above shows the total number of occurrence of each hardware and software events as well as the total elapsed time for running the program.

The other mode of an HPC-based measurement in the Linux perf is the *Sampling* mode in which the results not only contain the number of total events count, but also has the program execution profile. Access to a program execution profile allows one to easily identify the time-consuming function(s) of the programs that is the main goal of almost every performance analysis project.

The Linux Perf tool in the sampling mode provides `-c` and `-F` flags to specify the sampling period and the sampling frequency respectively. If the sampling frequency is not

provided, then it uses a sampling frequency of 4,000 Hz for all of the hardware events. In Section 1.7.2, we investigate the impact of the sampling period and frequency on the accuracy and the overhead of the `perf_event` sub-system.

To run the Perf in sampling mode we need to use `perf record` sub-command that captures a profile of the program during its execution and stores the results in a single binary file. After capturing the results, we can use the `perf report` sub-command for parsing the resutls and generating reports. The following shows the `perf record` command that runs the sleep commands:

```
$ perf record sleep 5
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.017 MB perf.data (7 samples) ]
```

It took seven samples and wrote all of the information and the captured profile of the program's execution in a single "perf.data" binary file. Also, the second line shows that the `perf_event` sub-system woke up the Perf tool only one time to read the samples from the memory mapped pages and write them into the `perf.data` file.

The Perf tool provides an easy-to-use command called `perf report` for parsing the `perf.data` files. The results below show the approximate total cycle events count alongside a sorted list of the hottest symbols (functions).

```
$ perf report --stdio
# Total Lost Samples: 0
#
# Samples: 7  of event 'cycles'
# Event count (approx.): 3157333
#
# Overhead  Command  Shared Object     Symbol
# ........  .......  .................  ........................
#
    91.22%  sleep    [kernel.vmlinux]  [k] unlock_page
     8.49%  sleep    [kernel.vmlinux]  [k] __inode_permission
     0.28%  sleep    [kernel.vmlinux]  [k] native_write_msr_safe
#
# (For a higher level overview, try: perf report --sort comm,dso)
#
```

Another flag that is available on both `perf record` and `perf stat` tools is the `-e` flag that enables one to specify the name of the event(s) that we want to monitor. The `perf list` sub-command shows a list of available hardware and software events on the system. In the following example we use `-e` flag to monitor the total number of taken branches while executing the `sleep` command. Moreover, in some architectures such as x86[1] we can specify a modifier for each event by appending them to the event name. Using modifiers enables us to distinguish between the events generated in different CPU privilege levels. For instance, we can only capture the events that occur in the kernel space. In Table 1.2, a complete list of the modifiers that can be used in conjunction with each hardware event is shown.

Modifiers	**Description**	**Example**
u	monitor at privilege level 3, 2, 1 (user)	event:u
k	monitor at privilege level 0 (kernel)	event:k
h	monitor hypervisor events on a virtualization environment	event:h
H	monitor host machine on a virtualization environment	event:H
G	monitor guest machine on a virtualization environment	event:G

Table 1.2: Perf events modifiers

In the example below, we run the `perf stat` sub-command with `:u` modifier to count the number of taken branches and retired instructions[2] that occur in the user-space during the execution of the `sleep` program.

```
$ perf stat -e branches:u sleep 5

 Performance counter stats for 'sleep 5':

           47,026      branches:u
          209,787      instructions:u

       5.000792210 seconds time elapsed
```

The sampling options can be enabled through the `sample_type` field. For instance, in the `perf record` program we can generate a full call graph of the program's execution by providing the `-g` flag. However, storing more information in each sample will introduce more overhead. The list of avaiable sampling options is shown in Table 1.3.

[1] ARM CPUs do not support event's modifiers.

[2] Instructions that are actually executed on the CPU and their results are written back to the CPU registers.

Flag	Description
PERF_SAMPLE_IP	Instruction Pointer
PERF_SAMPLE_TID	Process or thread ID
PERF_SAMPLE_TIME	Sample Timestamp
PERF_SAMPLE_ID	Unique ID of the opened event
PERF_SAMPLE_CPU	CPU number
PERF_SAMPLE_PERIOD	Latest sampling period
PERF_SAMPLE_IDENTIFIER	Placing SAMPLE_IP in a fixed location in raw data

Table 1.3: List of Sampling Options

If we specify more events than there are actual hardware counters, the kernel uses a time-based multiplexing method to give each hardware event a chance to access to the hardware counters.

```
$ perf stat -e branches:u,branches:u,branches:u,branches:u,branches:u ./test

 Performance counter stats for './test':

 9,693,894      branches:u                    (75.31%)
10,634,778      branches:u                    (82.51%)
10,159,684      branches:u                    (83.45%)
 9,679,721      branches:u                    (83.67%)
 9,868,784      branches:u                    (75.21%)

    0.038256859 seconds time elapsed
```

It is important to understand that in multiplexing mode, an event is not measured all the time as the hardware counters are shared among all of the monitored events. Hence, at the end of the execution of the program, Perf tool scales up the results based on the `total_time_running` and `total_time_enabled` values that can be accessed through the `perf_event` file descriptor. Formula 1.5 will be used to calculate the total number of events in the final report.

$$estimated\,count = raw\,count \times \frac{total\_time\_enabled}{total\_time\_running} \tag{1.2}$$

As an example, let's assume that our CPU has N physical hardware counters and we want to measure M hardware events in which $M > N$. Since the number of hardware events that we need to count is more that the number of the available hardware counters, the PMU will share the hardware counters among the requested events in a round-robin fashion to give every event a chance to be measured on a real hardware counter. Also, for each hardware event the kernel stores the total time that the event is acutually measured on the hardware counters as well as the total number of events count. At the end of the measurement, the kernel uses the Formula 1.5 to compute the total event count estimation for each measured event. If we do not consider the hardware counter switching overhead, then $\frac{total\_time\_enabled}{total\_time\_running}$ will be equal to the $\frac{M}{N}$.

In a situation in which an event did not get a chance to access the hardware counter *total_time_running* will be equal to zero and the Perf reports "`not counted`" in the output for that particular event. These two values will be provided in user-space upon a read on the `perf_event` file descriptor if `perf_event_attr.read_format` is configured with `PERF_FORMAT_ TOTAL_TIME_ENABLED` and `PERF_FORMAT_TOTAL_TIME_RUNNING` at the time of creating the event using `perf_event_open` system call.

It is always good to know the maximum number of hardware performance counters available on the CPU to prevent subsequent scaling and inaccurate results. The number of hardware performance counters in the common processors is provided in the Table 1.4.

So far, we briefly explained the Perf subcommands that we need to know for conducting the HPC-based measurements. In the rest of this section, we will evaluate the accuracy of the Perf tool by performing a set of measurement based comparisons in both *Counting* and *Sampling* modes on different architectures [2, 3, 10, 11].

1.6 Evaluation Methods

In order to evaluate the accuracy of the Perf results in this section, we first explore alternatives to hardware performance counters. Next, we conduct a series of experiments to compare and contrast the results of Perf with other methods in order to better assess accuracy.

To evaluate the accuracy of a given tool, it is necessary to find other methods that are, comparatively speaking, reliable. Accordingly, we use the following three methods:

- Event count estimation using static program analysis (mathematical model)

Processor	Hardware Counters
Intel Pentium 4	18
Intel Nahelem	11
Intel IvyBridge	11
Intel SandyBridge	11
Intel Atom	7
Intel Core 2 Duo	5
Intel Pentium III	5
intel Pentium Mobile	2
AMD Athlon	4
AMD G-Series	4
ARM Cortex-R4	3
ARM Cortex-A5	2
ARM Cortex-A8	4
ARM Cortex-A9	6
ARM Cortex-A53	6
POWER4	8
SPARC	2
MIPS 1004K	2
MIPS 74K	4
PowerPC	4

Table 1.4: Number of hardware counters on different CPUs

- Binary instrumentation methods
- Hardware traces

In the following three sections, we briefly explain these methods and the preferred tools for performing the measurements.

1.6.1 Event Count Estimation using Static Analysis

In this method, we count the total number of occurrence of each event without actually running the program; instead, we estimate the event counts based on the information that we obtain from the program source code. The limitations of this method make it

near impossible to model the occurrence pattern of some low-level events such as CPU L1 cache misses, TLB cache misses or even CPU cycles as we need to model the entire CPU components.

We use this method to model the retired instructions and taken branches events as they are easily modeled mathematically. It is important to note that, in this method, we do not make any measurements; instead, we only count the number of instructions and branches based on our assumptions about CPU concepts and their operations. Also, for counting the retired instructions events, we need to analyze the source code of the C program in assembly level.

Providing "`-S`" flag in GCC compilation command will generate a file that contains the generated assembly code of instructions for a given C source code.

```
$ gcc my_program.c -O0 -S
```

To better understand the program execution flow at the assembly level from the beginning of a program (`_start` entry function) execution to the end (`exit` function), we can use the *objdump* Linux tool. The *objdump* tool disassembles the executable files and extracts the actual instructions that will be executed from the beginning of the program to the end. In the following example, the `-d` flag is used for extracting the contents of the executable sections.

```
$ objdump -d ./program
```

We use a simple microbenchmark program to analyze the accuracy of Perf in counting the number of retired instructions and taken branches in a user level by employing the following mathematical method as a baseline.

```
1  #include <stdlib.h>
2
3  #define MILLION 1000000
4  #define KILO  1000
5
6  int main(int argc, char **argv){
7    int loop_count = argc > 1 ? strtoul(argv[1], NULL, 0) : 1;
8    int coef = argc > 2 ? ((argv[2][0] == 'm') ? MILLION : ((argv[2][0] == 'k'
       ) ? KILO : 1)) : 1;
9    int long iterations = coef * loop_count;
10   for(long i = 0; i < iterations; i++){
11     __asm__("NOP"); //no operation
```

```
12    }
13 }
```

Snippet 1.2: *noploop* Micro benchmark program

The `noploop` program takes two optional parameters. The first parameter indicates the number of loop iterations and the second parameters will multiply loop iteration by 10^6 (million) in case of providing m or 10^3 (kilo) in case of k.

In the `noploop` program, we expect to have one *taken branch* per each loop iteration. Also, breaking down the code to the assembly level can tell us how many retired instructions should be executed on each iteration of the loop. For counting the number of cycles consumed on each loop iteration, we will use other low-level methods such as hardware traces.

The Snippet 1.3 shows the basic block of the main loop of the `noploop` microbenchmark program (x86-64 machine code).

```
1 .L5:
2   NOP
3   addq   $1, -8(%rbp)
4 .L4:
5   movq   -8(%rbp), %rax
6   cmpq   -24(%rbp), %rax
7   jl    .L5
```

Snippet 1.3: *noploop* Basic block instructions

Therefore, on each iteration of the loop basic block, *five* instructions must be executed on the machines that use x86-64 Instruction Set Architecture (ISA). This number varies among the different machines as they might use different ISAs. It is important to note that the Perf tool counts all retired instructions and taken branches that occur during the complete execution of the program. Since we are only interested in counting the events that occur during the execution of the loop basic block, the Perf's results will be subject to an overcount (because of counting unrelated events) when we compare it to the computed values that we obtained from the static analysis method. To eliminate the effect of overcounting, we first count the number of instructions and branches that are captured from the *empty* C program. Next we subtract the total event counts from the results captured from the original `noploop` program.

1.6.2 Dynamic Binary Instrumentation

Dynamic Binary Instrumentation (DBI) is a technique that injects instrumentation codes into the binary executable files to collect the desired information from the program at the run time. The level of binary instrumentation has a significant impact on the overhead imposed by the DBI tool. For example, if we want to count the number of executed instructions using a DBI method, we need to instrument every single instruction of the program that results in enormous overhead. However, in our case, we should not be worried about the overhead induced by the instrumentation as we only need to use a DBI tool verify the results the Linux Perf.

We select Pin, a dynamic binary instrumentation framework from Intel, that supports Linux, Windows, and Android operating systems, as our DBI tool. The only limitation that we have in using the Pin is that we need to limit our experiments to the x86-64 and IA-32 platforms. However, it gives us a freedom to select the real benchmark programs instead of using microbenchmarks for conducting the instruction counts experiments (This option was not available in a model-based analysis). Also, since DBI tools such as Pin only instrument user-space programs, we would not be able to count the instructions that are executed in the kernel space. For instance, if our program invokes a system call, the execution of the program will be switched to the kernel space that results in not counting the instructions that are executed in the kernel space. Therefore, to prevent any measurement error, we use "`:u`" modifier to force the `perf_event` to only measure events that occur in the user-space.

To perform a DBI-based measurement on our microbenchmark program, we use the available `icount` and `jumpmix` Pin tools that are reside in the `source/tools/SimpleExamples` directory. In Section 1.7.2, we evaluate the determinism of Perf's results in counting the number of taken branches and retired instructions events by employing the results of Pin as a reference.

1.6.3 Hardware Traces

Hardware traces are relatively new technology that enable the capturing of hardware events in real-time with zero overhead. Having no overhead makes it a fascinating tool for conducting a measurement-based performance analysis. Hardware trace is also a good approach for those who need to analyze the behavior of their hard real-time programs in which missing a deadline causes a total system failure. Most of the time, the performance measurement in real-time environments needs a level of granularity that is often not provided by the traditional debugging and profiling tools. Hardware traces are the best solution for a situ-

ation in which a real-time program missed a deadline because of a subtle performance bug that cannot be easily identified by using the traditional debugging methods.

Hardware traces on ARM microprocessors are provided with the aid of CoreSight architecture. *CoreSight* architecture provides a real-time, on-chip, tracing and debugging solution with respect to the following specifications [1].

- Supporting on-chip trace debug
- Tracing and debugging multi-core system
- Compatibility of the components from different vendors
- High bandwidth trace collection from multiple sources
- Non-intrusive access to the tracing and debugging components
- Attaching to the running target without performing any software or hardware reset
- Supporting embedded (internal) and external trace buffers
- Controlling access to the debug and trace functionality in the hardware level
- Capturing the trace for a large period and storing them on the external trace buffers

The ARM CoreSight Trace Macrocell offers the following solutions for a non-intrusive program tracing across a System-On-Chip (SoC).

- *PTM*: PTM provides a real-time and cycle accurate instruction level traces from the Cortex-A9 processor with zero overhead.
- *Embedded Trace Macrocell (ETM)*: ETM is a non-intrusive and cycle accurate program and data access traces.
- *Instrumentation Trace Macrocell (ITM)*: ITM provides a high-level view through the *instrumentation* in contrast to ETM and PTM trace sources that only provide a low-level trace view. ITM trace source is only available on ARM Cortex-M processors.

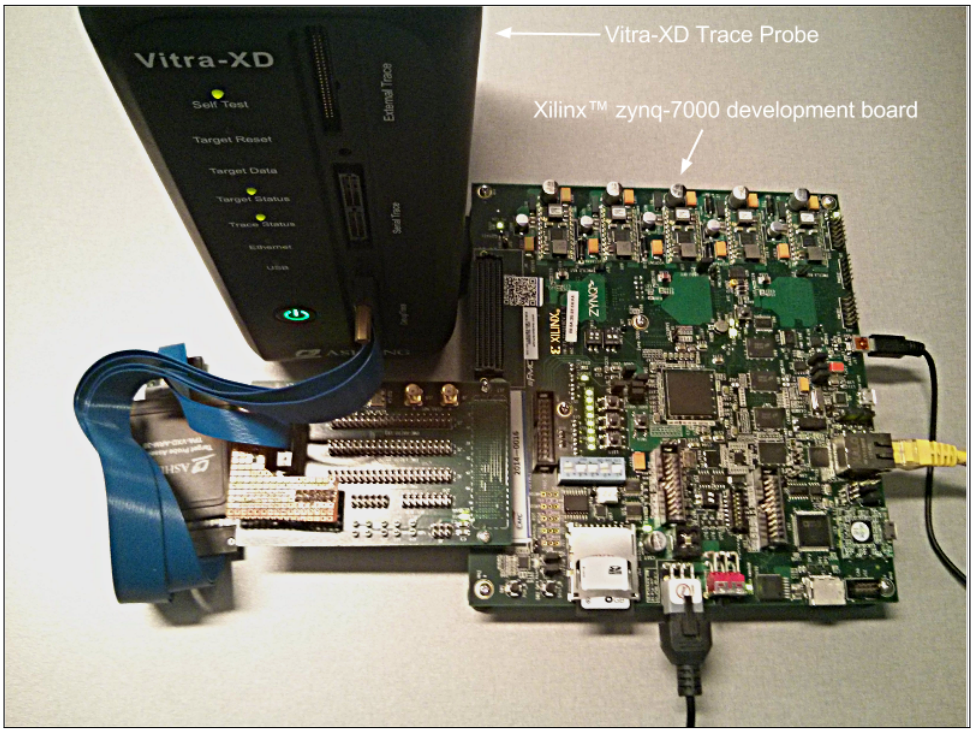

Figure 1.5: Ashling Vitra-XD setup

We use Xilinx Zynq-7000 All Programmable SoC[16] that integrates a Dual core ARM Cortex-A9 processing system with an on-chip FPGA as our target development board. We have made some modifications to the both software (FPGA) and hardware to bring out the PTM traces from the processing cores. Figures 1.5, 1.6 show the hardware experiment setup for the Ashling Vitra-XD and the ARM DSTREAM trace probes.

We use the ARM DSTREAM and the Ashling Vitra-XD trace probes that are specifically designed to capture the hardware traces from the ARM processors. Both of them are capable of working with ETM and PTM trace modules on ARM processors. PTM generates the instruction traces based on the Program Flow Trace (PFT) architecture. Also, it only generates the trace at certain points of the program execution flow, called *waypoints*, that includes branches, exceptions and processor state change events to reduce the size of the trace and prevent FIFO overflow. Trace tools use waypoints in conjunction with a copy of the compiled program (with the debug information) to reconstruct the full execution flow of the program. Also, PTM can be configured in a way to report the cycles count between the two waypoints as well as the timestamp of each trace data. Figure 1.7 shows

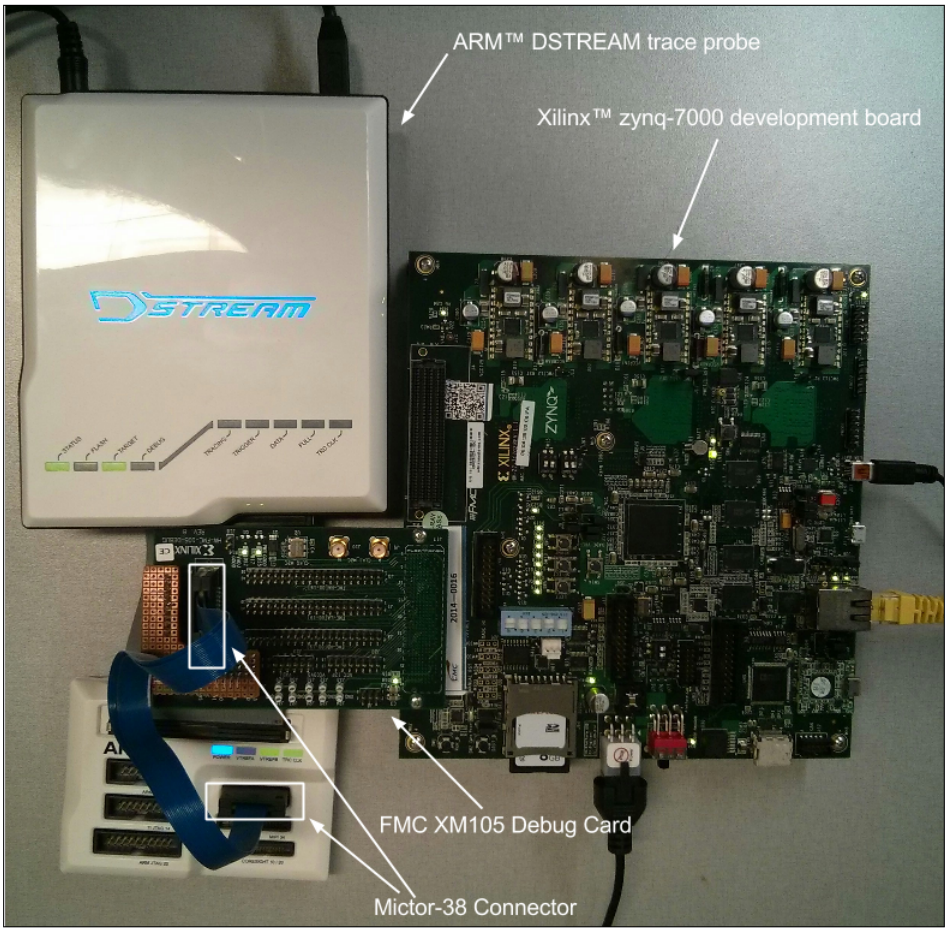

Figure 1.6: ARM DSTREAM setup

the raw trace format captured from the Ashling Vitra-XD trace probe that is extended by the CPU core ID and the high-resolution timestamp [3]

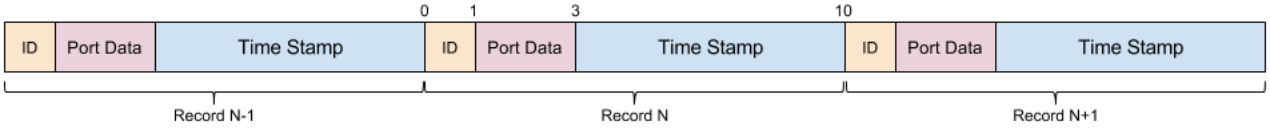

Figure 1.7: PTM Raw Trace Record Format

[3]This output format is not a standard output format for the PTM trace. The standard format only contains *Port Data* field.

The PTM raw trace data consists of the PTM packets that are represented on the records. The length of the packets is not fixed and depends on the type of the packet. The header of each packet starts with a special value of the *Port Data* field. Based on the type of the packet and what is included in the packet, the size of the payload will be calculated. We have eleven types of the packets in total that are defined in the PFT architecture that are assembled in the PTM raw trace. Table 1.5 shows a list of the PFT packets and a short description of each one.

Packet Type	**Header Format**	**Category**	**Discription**
Atom	0b1xxxxxx0	Instruction	Information about taken or not taken branch in the waypoint (and cycle count if enabled)
I-sync	0b00001000	Sync.	Generated to help the trace reconstructor to synchronize the instruction address, Context ID and security state
A-sync	0b00000000	Sync.	Generated when PTM is enabled or reprogrammed (Alignment Sync.)
Waypoint Update	0b01110010	Instruction	PTM generates waypoint packet when a non-deterministic branch instruction executed
Branch Address	0bCxxxxxx1	Instruction	Indicating a change in the program flow and the new IP address
Trigger	0b00001100	Misc.	Reporting an important PTM hardware events
Context ID	0b01101110	Instruction	Generated when the context ID register is changed (context switch)
VMID	0b00111100	Instruction	Reported when Virtual Machine ID is changed (if the processor has Virtualization Extensions)
Timestamp	0b01000x10	Sync.	Provides processor core specific timestamp on a multi-core environment
Exception return	0b01110110	Instruction	Indicates that the processor returns from an execption handler
Ignore	0b01100110	Misc.	It has no effect and is only inserted when there are no sufficient trace to fill the data trace port

Table 1.5: PFT Packets Format

As we mentioned earlier in Section 1.1, ARM tried to overcome poor software visibility of the hardware traces by including the value of the "Context ID" register in the Context ID packet of the PTM trace. On ARM machines, *c13* or *Context ID* register is one of the system control coprocessor registers that keeps the Address Space Identifier (ASID)[4] and the current process ID. On the Linux operating system, we must compile the kernel with the

[4]ARM based processors use ASID in the TLB cache to prevent a TLB flush after each context switch

CONFIG_PID_IN_CONTEXTIDR=y option to force the kernel to write the PID of the executing process into the *Context ID* register. Although process trace filtering is made possible using the Context ID packet in a raw trace data, we still cannot differentiate between the traces that are generated from the different threads that are living in a process. The current version of the ARM DS-5 development studio does not support process trace filtering in the hardware level. However, Ashling provides a hardware level solution to allow the users to only capture the hardware traces of a specific process on the Linux operating system (Context ID filtering is implemented in the hardware).

In this work, we take the advantage of the cycle accuracy of the PTM trace to verify the results of the Linux Perf tool in counting the number of CPU cycle events. Figure 1.8 shows the total number of cycles that are consumed between the two waypoints that is equal to the total number of cycles in each iteration of the main loop.

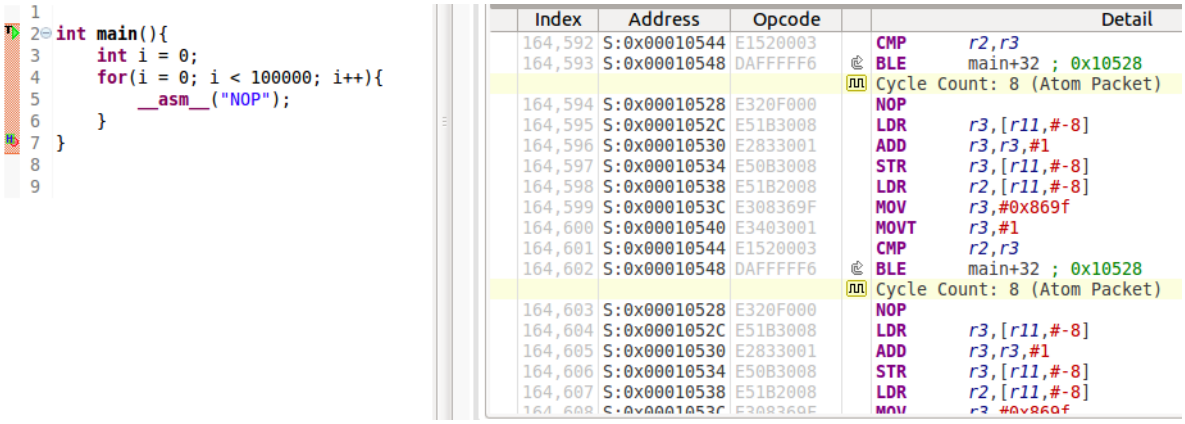

Figure 1.8: Main loop cycles count using PTM trace on ARM DS-5 development tools

In the next section, we will verify the results of the Linux Perf tool in counting hardware events such as CPU cycles, retired instructions, and taken branches by conducting a set of measurement-based experiments. For each event, we employ one or more measurement methods as the baseline.

1.7 Evaluation

This section is dedicated to evaluating the Linux Perf tool based on its accuracy, determinism and overhead. All the experiments have been performed on the Datamill open

source benchmarking infrastructure to provide the most accurate results [9]. The software configuration of the machines that we used in our experiments are listed in Table 1.6.

Architecture	Linux Version	Compiler	Perf Version
x86_64	3.18.11-gentoo	GCC 4.8.4	3.12
i686	3.18.11-gentoo	GCC 4.8.4	3.12
armv7l	3.15.0-rc8	GCC 4.8.3	3.12

Table 1.6: Datamill Machines Configuration

1.7.1 Determinism

The ideal hardware performance counter in *counting* mode should provide a run-to-run consistent results. However, our experiments reveals that the Linux Perf tool in counting mode has failed to provide such deterministic results across most of the tested machines. We ran the following "`perf stat`" command ten times and calculated the *Median* and *Mean Absolute Deviation (MAD)*.

```
perf stat -e <event name>:u ./noploop 10 m
```

We use *taken branch* and *instruction* events in conjunction with the `:u` modifier to only count the user-space events. The experiment results are shown in Table 1.8.

	Taken Branches		Retired Instructions	
Machine	**Median**	**MAD**	**Median**	**MAD**
AMD G-T56N	10,018,237	0.48	50,092,625	0.88
Intel Atom	10,018,750	0.72	50,094,353	0.42
Intel Core i5-2500	10,018,717	0.54	50,093,050	0.88
Intel Core i5-4300U	10,018,508	0.54	50,093,570	0.32
Intel Pentium 4	10,019,198	0	50,094,831	3082.56
Intel Pentium M	10,018,365	2.84	50,094,741	0.84
Intel Xeon	10,018,469	2.84	50,093,424	2.30

Table 1.8: Perf Determinism in Counting Mode

The only event count that we found deterministic is the *Taken Branch* event on *Intel Pentium 4* CPU. However, on the same machine the *Retired Instruction* event suffers a significant deviation in counting results.

Indeterministic results are not limited to the Linux Perf and the underlying perf_event sub-system as Vincent M. Weaver *et al.* reported the same results on *Perfmon2* [41]. Therefore, we can conclude that the root cause of the indeterministic results is the underlying processor PMU that needs to be redesigned.

1.7.2 Accuracy

To evaluate the accuracy of the Perf's results on `x86_64` and `i686` architectures, we use the results from the Pin as a baseline. However, on ARM machines, we need to employ other approaches due to the ARM's PMU limitations in counting the user-space hardware events. Hence, on `armv7l` machines we evaluate the results of Perf against the estimated value that we will compute using the mathematical models. Also, we use the PTM hardware traces to calculate an estimation of the total CPU cycles count.

Tables 1.10 and 2.4 show the results of the experiments which are performed on seven `x86_64` and `i686` machines that are running on the Datamill. We use `noploop` as a microbenchmark program and the `Coremark` as a real benchmark for this experiment.

	Taken Branches			**Retired Instructions**		
Machine	**Perf**	**Pin**	**Diff.**	**Perf**	**Pin**	**Diff.**
AMD G-T56N	10,018,257	10,018,213	-44	50,092,692	50,095,305	2,613
Intel Atom	10,018,769	10,018,728	-41	50,094,420	50,097,036	2,616
Intel Core i5-2500	10,018,738	10,018,696	-42	50,093,117	50,095,734	2,617
Intel Core i5-4300U	10,018,527	10,018,488	-39	50,093,638	50,096,255	2,617
Intel Pentium 4	10,019,198	10,018,570	-628	50,127,682	50,094,831	-32,851
Intel Pentium M	10,018,365	10,018,225	-140	50,094,232	50,094,739	507
Intel Xeon	10,018,482	10,018,444	-38	50,093,481	50,096,099	2,618

Table 1.10: Perf vs. Pin - noploop Microbenchmark

The results from Tables 1.10 and 2.4 indicate a quite small difference between the results of Pin and Perf (in *counting mode*) with the exception of *Intel Pentium 4*. On the *Intel Pentium 4* machine we observe an average error of 3% and 47% when counting the taken branches and retired instructions events, respectively.

	Taken Branches			Retired Instructions		
Machine	**Perf**	**Pin**	**Diff.**	**Perf**	**Pin**	**Diff.**
AMD G-T56N	770,861,452	770,861,330	-122	3,575,229,680	3,575,238,830	9,150
Intel Atom	770,861,970	770,861,846	-124	3,575,231,447	3,575,240,536	9,089
Intel Core i5-2500	770,861,969	770,861,764	-205	3,575,232,943	3,575,241,116	8,173
Intel Core i5-4300U	770,861,684	770,861,475	-209	3,575,230,882	3,575,239,088	8,206
Intel Pentium 4	747,147,835	762,483,700	15,335,865	5,368,338,190	3,518,532,826	-1,849,805,364
Intel Pentium M	765,904,479	765,903,761	-718	3,571,158,690	3,571,160,844	2,154
Intel Xeon	770,861,551	770,861,348	-203	3,575,230,446	3,575,238,631	8,185

Table 1.12: Perf vs. Pin - Coremark Benchmark

We have selected the Zynq-7000 platform which uses the ARM Cortex-A9 processor to evaluate the reliability and accuracy of the Linux Perf tool and the underlying PMU of the ARM processor. We employed the static program analysis method by providing a mathematical model to estimate the total number of taken branches and retired instructions events. Also, we used the PTM hardware trace to evaluate cycles event count. As we mentioned earlier in Section 1.5, since the ARM processor is not able to distinguish between the kernel and user events, we use two microbenchmark programs, namely `noploop` and `noloop`, for estimating the total number of events generated in the mail loop. Hence, to count the number of events that occurred in the *for loop*, we only need to subtract the hardware event counts from the two microbenchmarks. Table 1.13 shows the results of the experiment we ran on the Zynq-7000 development board. The expected value for each event is calculated based on the information we have extracted from the instruction level source code analysis, hardware traces and the mathematical model of the main *for loop*.

Event Name	`noploop` **Count**	`noloop` **Count**	**Diff.**	**Estimation**	**Percent Error**
Retired Instruction	80,752,482	517,459	80,235,023	80,000,000	0.29%
Taken Branches	10,075,123	49,578	10,025,545	10,000,000	0.25%
Cycles	81,849,214	1,506,682	80,342,532	80,000,000	0.42%

Table 1.13: Perf's Accuracy Evaluation on ARM Cortex-A9

Another source of measurement error that we could not eliminate is the context switch(es) that may occur during the execution of the *for loop* basic block of the `noploop` program. Therefore, the actual error might be less than the number we included in Table 1.13. Despite the measurement error that we could not eliminate from our experiment, the Perf's event counts result is quite close to our estimation.

As previously stated, the total number of events reported by the `perf record` is not as accurate as the *counting* mode (`perf stat`). However, our experiments show that in long running programs (∼400 billion instructions) the *relative error* is small enough that the accuracy of the reported number of the events is no concern. The results of the execution of the `Coremark` benchmark program using the `perf record` sub-command with the default options are shown in Figure 1.9. We use the results of the Perf in counting mode as a baseline for comparision (The relative error is multiplied by 10,000).

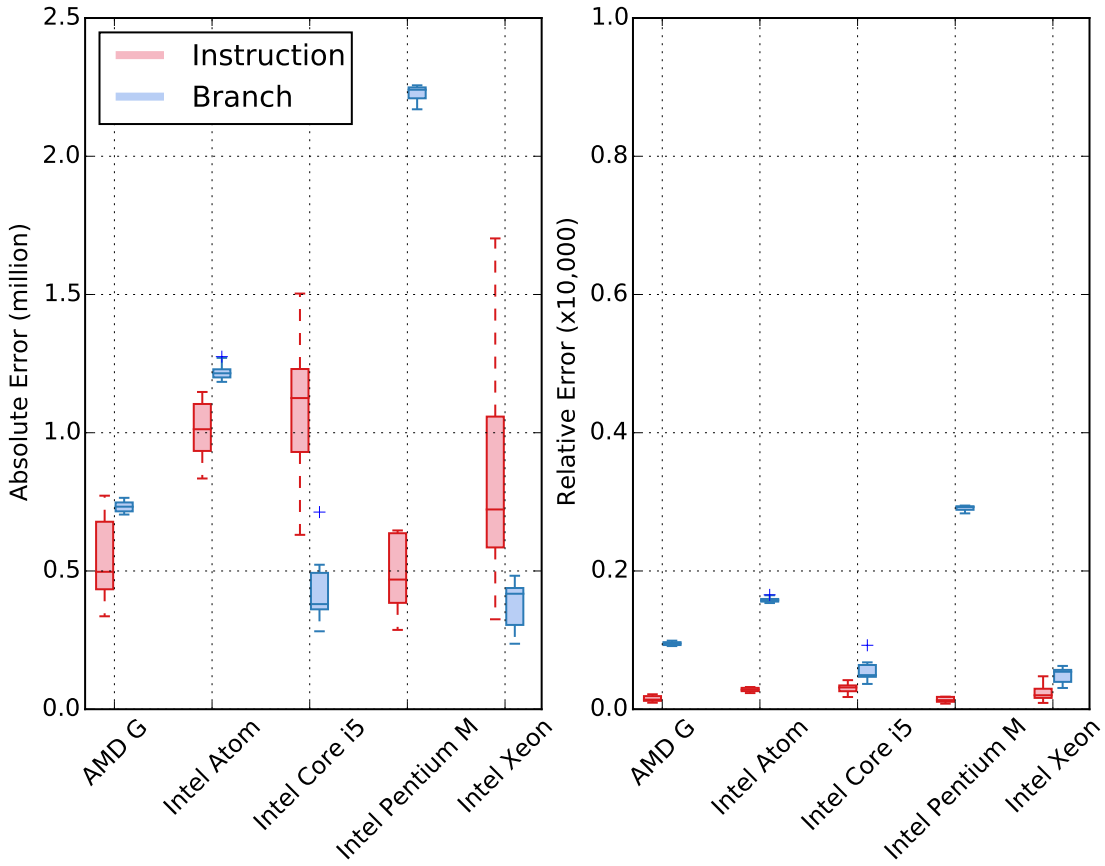

Figure 1.9: Accuracy of Perf in sampling mode (default settings)

There is a trade-off between the sampling granularity and the total counted events. Therefore, a special care must be taken when one changes the *sampling frequency* or the *sampling period* of the `perf record` for achieving a better granularity. In Equation 1.3 we summarize the asymptotic relations between the factors in sampling mode.

$$Granularity \sim Sampling\ Frequency \sim Error \sim Overhead \sim \frac{1}{Sampling\ Period} \tag{1.3}$$

Since we did not provide any parameter for the `perf record` program, it uses the default sampling rate which is hard-coded to the 4,000 Hz. Put differently, the kernel (perf_event) automatically adjusts the sampling period to record 4,000 samples per second. We can change the sampling rate on the `perf record` by providing "`-F`" or "`--freq`" arguments. Also, "`-c`" or "`--count`" arguments can force the kernel to use a *fixed* sampling period. The Figure 1.10 shows the effect of changing the *sampling frequency* on the *sampling period* and the total number of captured samples.

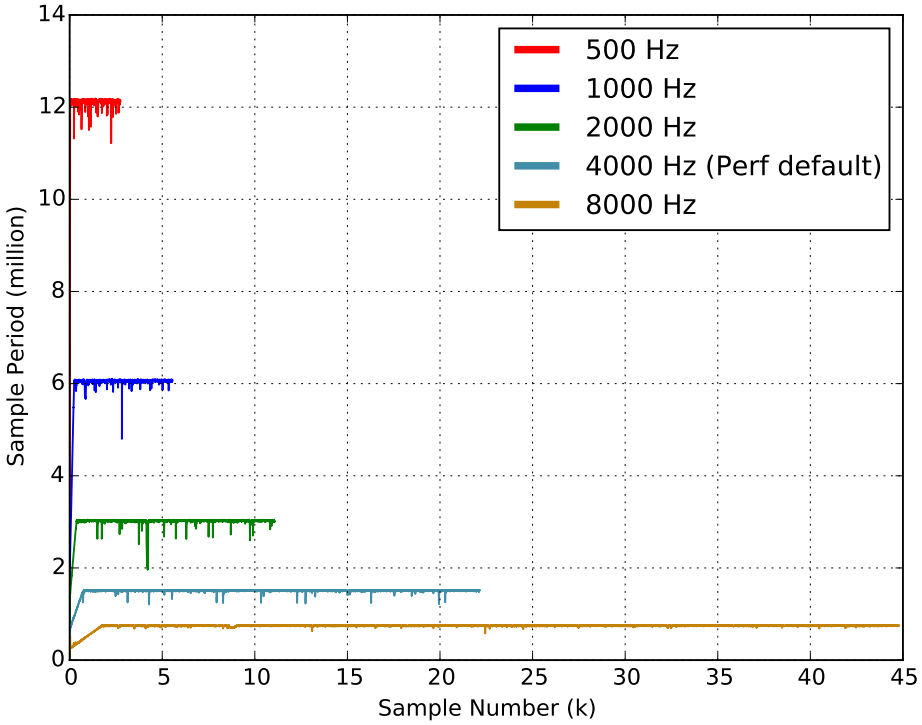

Figure 1.10: Effect of sampling frequency on sampling period

1.7.3 Overhead

Overhead is one of the most important factors to consider when selecting a method for executing measurement-based performance analysis. The hardware performance counter method is classified as a low-overhead approach since the performance-sensitive aspects are already implemented in the hardware. We could not observe significant overhead in running the Perf in counting mode as it only executes *start*|*stop*|*read* instructions for counting the total number of the events. However, in the sampling mode both perf_event in the kernel space and the Perf tool in user-space add a run-time overhead to the sample-based profiling system. In sampling mode, the PMU periodically interrupts the kernel to record a new sample, and, on the other hand, the perf_event occasionally wakes up the Perf tool to save the samples that are stored in the ring-buffer. In Figure 1.11 we visualize the overhead of the `perf record` for *Cycles*, *Instructions* and *Branches* hardware events on different machines.

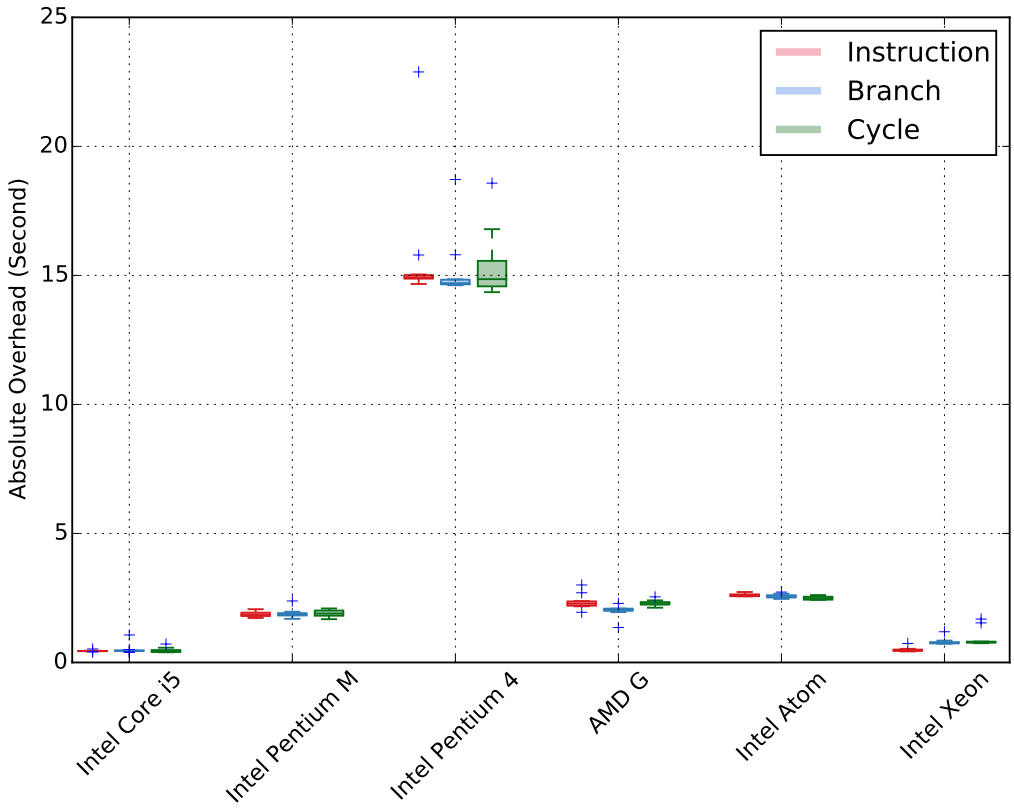

Figure 1.11: `perf record` Run-time Overhead

Our experiment shows 5.92% relative overhead on the *Intel Pentium 4* machine. Another observation that we can extract from this experiment is that the type of hardware event does not have a significant impact on the overall run-time overhead. This correlation can be explained by looking at the relatively equivalent number of taken samples for each hardware event that are only affected by the sampling rate (not the hardware event type). Therefore, we designed another experiment to find the impact factor of sampling frequency on the overhead. The results of the experiment are visualized in Figure 1.12.

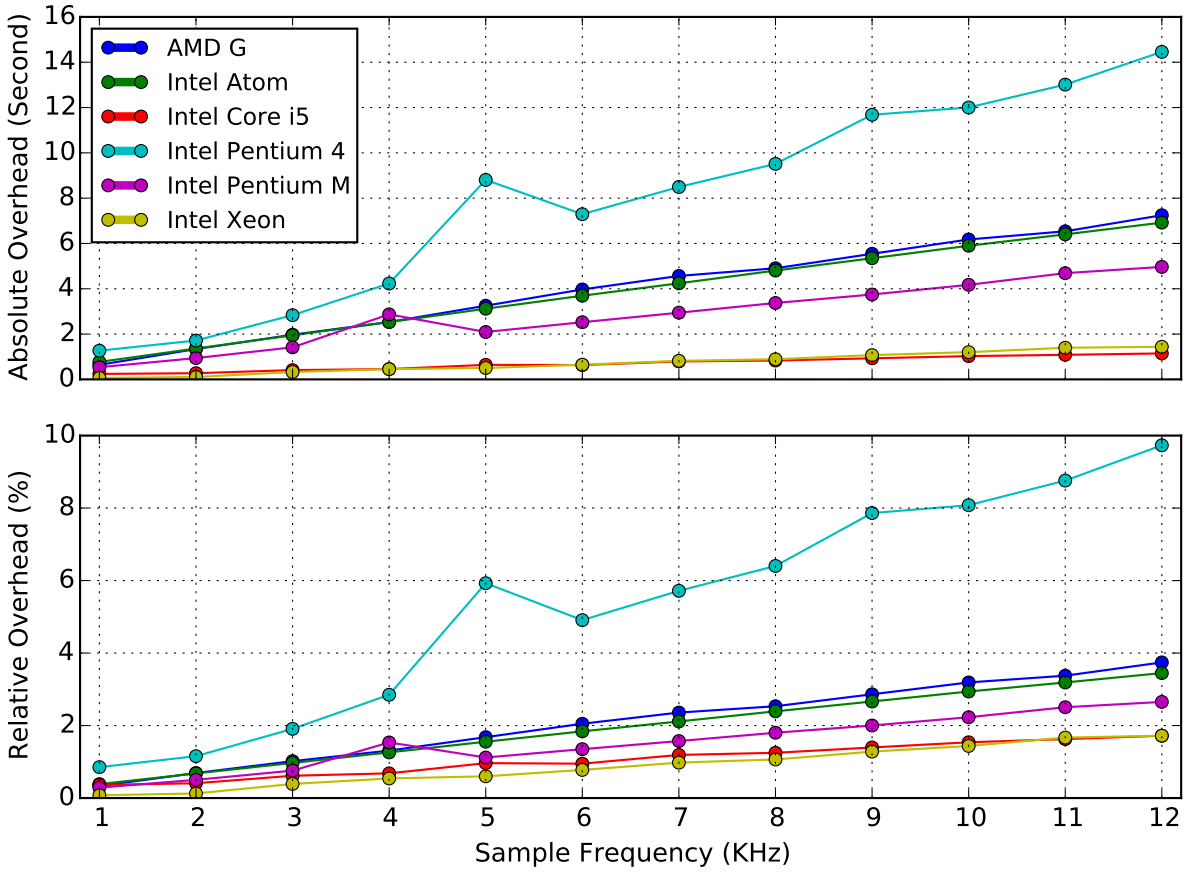

Figure 1.12: Effect of Sampling Rate on Overhead

Figure 1.12 shows 9.8% error in average at the frequency of 12 KHz in *Intel Pentium 4* machine. However, on the same machine, at the frequency sampling of 4 KHz, or the default sampling rate of the `perf record`, we observe 2.5% relative error. In conclusion,

our results show that the overall overhead does not increase *significantly* with an increase in sampling frequency (except *Intel Pentium 4*).

1.8 Lessons Learned

Access to the `perf_event` sub-system is provided through the `perf_event_open` system call that enables users to interact directly with the hardware performance counter. However, during our experiments, we realized that the implementation in which we interact with the `perf_event` sub-system has a significant impact on the accuracy and the overhead of the measurement. In *counting* mode, we are limited to a *start*|*stop*|*read* model that is associated with an negligible over-counting. In contrast, we have many tunable options and implementation models in the *sampling* mode. As we discussed in Section 1.4.2, the wakeup notification handling that indicates that the ring buffer is ready to be written by the program is one of the main sources of the overhead. We find that the *polling* method (over wakeup signal) that is used by the Linux Perf tool has much better performance over the *signal handling* method.

Hardware trace is an amazing technology that enables us to record a complete history of the program's execution. There are many challenges in working with this technology that limit its application to certain industry level projects. However, we introduced this technology to the academy as a unique tool for performance debugging and analysis. We began with the DS-5 debugger and ARM DSTREAM high-performance trace unit that were equipped with a 4 Gigabyte trace buffer. Considering the high-speed of ETM/PTM trace generation that is almost equal to the speed of the processor, we only could capture and store a few seconds of a program execution in the limited trace buffer. In addition to this limitation, the ARM DSTREAM trace probe did not provide a hardware level trace filtering in the Linux kernel debugging mode to *only* capture the ETM/PTM traces of the desired process. Therefore, we needed to take an extra post-processing step to filter out the unwanted traces of other processes that were running on the system at the time of capturing the hardware traces. Trace reconstruction was another challenge we faced while working with the hardware traces. Since the PTM raw trace format that we described in Section 1.6.3 is not in a an intellegible format, we need to perform a reconstruction procedure by matching the trace packets against the program executable file and the program source code. Unfortunately, the current sequential implementation of the raw trace reconstruction algorithm makes this process quite slow. For instance, in the ARM DS-5 debugger, downloading and re-constructing the PTM traces take about twelve hours for each Gigabyte of the hardware raw traces that are stored in the DSTREAM

internal buffer. We also have experience working with the Ashling Vitra-XD trace probe that uses Sourcery Code Bench as the frontend software. The Ashling Vitra-XD trace probe solves the trace buffer size limitation by employing a 500 GB trace buffer that enables one to capture up to *three hours* of continuous hardware traces. Also, in the Ashling solution, a process-based trace filtering is implemented in the hardware. This feature not only removes the burden of an extra post-processing step, but also safeguards the internal trace buffer from irrelevant process traces. The trace reconstruction on the Sourcery Code Bench software is also a tedious process that makes downloading the entire reconstructed trace to the host machine impractical. In the most recent version of the Ashling plugin for the Sourcery Code Bench platform, the designers have added a new feature that enables one to transfer the entire un-reconstructed data to the host machine. This add-on opens the door for future research and the possible applications of these valuable traces.

Chapter 2

Linux Kernel Binning Effect

2.1 Definition

"Binning" or "Bucketing" have different meanings in various contexts. In general, "Binning" is the act of placing items such as data into a fixed or variable set of containers called "Bins" or "Buckets". In computer software, we use this concept as a means of grouping data of the same structure. For example, in many filesystems the main memory is virtually divided into a fixed-sized (usually 4KB) blocks called memory page. Deviding memory into the fixed-size blocks helps the operating system to improve the performance by reducing the number of access requests to the memory and making the memory management more efficient. Despite the fact that binning enhances the throughput of the system by grouping relevant data into a set of buckets, sometimes it can have an adverse effect on the timing aspect of the individual sub-systems. In the rest of our thesis, we refer to *Binning* (in the Linux kernel) as a phenomenon in which the threshold of a particular bin (bucket) is reached that forces the kernel to take an appropriate action in response to that event that usually causes timing variability in the system.

2.2 Introduction

The Linux operating system understands binning in various levels such as hardware abstraction, resource management, and drivers. The concept of binning in Linux resource management components can be applied to the both time and memory resources. As an example of the time bins in the Linux kernel, we can refer to the scheduler time-slice known

as the Linux quantum that has a significant impact on the execution time of the programs. The Linux scheduler uses the timeslice to switch continuously between tasks and give every running program a chance to access the CPU. The time binning takes place in the kernel when the scheduler reclaims the CPU from a running process that already consumed its time slice.

Our focus in this work is on memory binning that always happens across the Linux kernel as it uses various types of bins for delivering the best throughput. A data buffer is one example of a construct that exhibits binning. A buffer can fill up with relatively little performance impact until a threshold is reached; at that point, and addition memory must be allocated. These "buckets" in the kernel include the following: buffers, stacks, queues, linked lists, and arrays. All of these buckets experience reallocation or resizing in the kernel. These buckets and data structures are heavily used in the kernel for a variety of different tasks, such as the `task_list` data structure that keeps the state of the running processes on the system.

In our research, we specifically investigate the impact of memory *binnig* on the execution time of Linux system calls that is imposed by the *Slab Allocation* memory management technique.

2.3 Slab Allocation

Linux kernel components and drivers frequently need to allocate memory for storing temporary objects such as `inode`, `task_struct` and `files_struct`. These small, fixed size objects are allocated and freed many times during the kernel's life cycle. In earlier implementations of the Linux kernel, it satisfied the requests for allocating and releasing these small objects through the `kmalloc` and `kfree` kernel functions that were initially optimized for the large physical memory allocations. Therefore, for the small, temporary objects that are often required by the kernel and drivers, the regular `kmalloc` and `kfree` allocation routines were inefficient, leaving the individual kernel drivers and modules to optimize their memory usage by themselves. One proposed solution was to create a global object cache in the kernel to isolate access to low-level page allocation and manage the kernel objects' allocation on behalf of the kernel components and drivers. In this method, each kernel component can create a private cache of a particular object type (C struct) and should make a request to the cache allocator for allocating the objects of the specified type. The cache allocator works in close collaboration with the memory management sub-system to preserve a balance between the needs for the memory of each driver or module and the system as a whole [19].

The Slab allocation is a memory management technique that is introduced in the Solaris 5.4 kernel [20]. Later on, other Unix and Unix-like operating systems such as Linux and FreeBSD integrated this technique into their kernel. The primary intention of using the slab allocation technique was to efficiently manage the allocation of the kernel objects and prevent memory fragmentation caused by memory allocation and deallocation. The kernel objects in this context mean the allocated and initialized objects of the same type that are usually represented in the form of *struct* in C programming language. These objects are only being used by the kernel core, modules, and drivers that run in the kernel space. Therefore, the access to these objects from the user-space programs is not possible unless the kernel provides an interface for accessing the content of these objects. For example, Linux uses `taskstat` struct to provide an interface to the user-space programs for accessing the relevant information of a running process (e.g., process ID, CPU time and major/minor page faults). The kernel uses slab allocation techniques to retain the allocated kernel objects of the same type upon subsequent requests of the same object. This action not only prevents excessive memory allocation and deallocation request but also avoids the overhead of initialization of each object. Sometimes in large and complex structures the cost of initialization of an object is more than the cost of memory allocation and deallocation, which significantly affects the performance of the kernel.

The following list is a partial list of the caches that are maintained by the kernel on our Linux machine (kernel version: 4.2.5). This information is provided from the `/proc/slabinfo` file (Reading needs a root access).

```
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab>
cifs_request           7      12   16512    1    8
cifs_inode_cache       0       0     736   22    4
inode_cache        12362   13888     568   28    4
dentry             68479   69132     192   21    1
iint_cache             0       0      72   56    1
buffer_head        84948   86112     104   39    1
vm_area_struct     48262   57178     184   22    1
mm_struct           1317    1512     896   36    8
kvm_async_pf           0       0     136   30    1
kvm_vcpu               0       0   16832    1    8
taskstats            144     144     328   24    2
task_struct          928    1089    3520    9    8
ext4_io_end         1456    1512      72   56    1
ext4_extent_status 16014   16014      40  102    1
jbd2_journal_handle  340     340      48   85    1
```

In the Linux kernel there are three different implementations of the slab allocation technique, namely *SLAB*, *SLUB* and *SLOB*. The Linux kernel can only use one of these implementations at a time. Therefore, we need to re-compile the kernel if we wish to change the slab allocator. In the next section, we explain the design philosophy of each implementation and enumerate their advantages and disadvantages.

2.4 Linux Slab Allocators

The first appearance of Slab allocators in Linux kernels dates back to 1996 when the first implementation of the slab allocation technique (Solaris type allocator) was added to the Linux kernel. Prior to that date, Linux uses the standard K&R heap allocator. In fact, the memory allocation for Linux kernel objects was equivalent to memory allocation for the user-space programs. Since the performance of the slab allocator sub-system has a significant effect on the overall performance of the kernel, developers continuously propose new methods to ameliorate current implementations. Figure 2.1 is a timeline that summarizes improvements to the Linux kernel slab allocator sub-system over the last two decades.

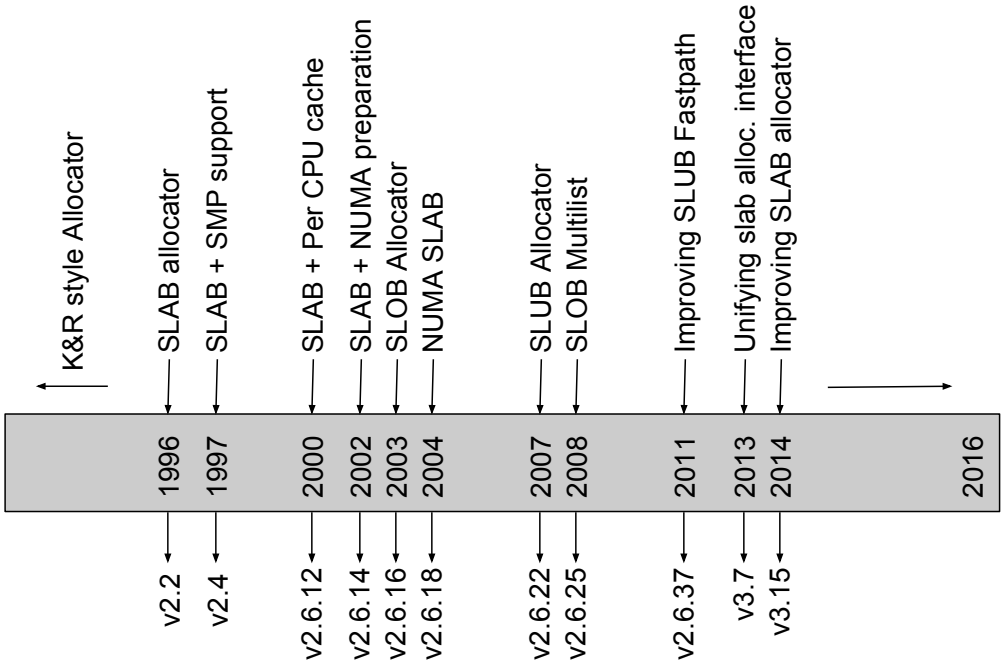

Figure 2.1: Slab Allocators Development Timeline

2.4.1 Understanding Slab Allocators

To understand Linux kernel slab allocators, we need to define the following terms, which appear frequently in the slab allocator source code.

- *Cache*: Cache is a group of the kernel objects of the same type. Cache is identified by a name that is usually the same as the C structure name. The kernel uses a doubly-linked list to link the created caches.
- *Slab*: slab is the contiguous chunk of memory is stored in one or more physical page(s) of the main memory. Each cache has a number of slabs that store the actual kernel objects of the same type.
- *Kernel Object*: The kernel object is the allocated and initialized instance of a C struct. Each slab may contain some objects (depending on the size of the slab and each object). A kernel object in the slab can be either "Active" or "Free."
  - Active: The object is being used by the kernel
  - Free: The object is in the memory pool and ready to be used upon request.

The Linux kernel slab allocation sub-system provides a general interface for creating and destroying a memory cache regardless of the type of slab allocator. These interfaces are defined in the `mm/slab_common.c` file.

- *kmem_cache_create*: `kmem_cache_create` enables us to create a new memory cache. This function allows five parameters:
  - *name*: A unique string as an identifier
  - *size*: The size of the object that will be stored in the cache
  - *align*: Object cache allignment
  - *flgas*: SLAB flags
  - *constructor*: The constructor function that will be called for initializing after object allocation

  `kmem_cache_create` returns an object of type `kmem_cache` that contains all of the parameters listed above and also a pointer to the first slab.

- *kmem_cache_destroy*: This function allows one to destroy a memory cache by providing the `kmem_cache` object of the desired cache.

SLOB, SLAB and SLUB allocators provide two functions for allocating (taking from cache) and freeing (putting back into the cache) a kernel object.

- *kmem_cache_alloc*: Allocate an object of a specific type from a cache (a cache for that specific object must be created before allocation). This function accepts the following parameters.
  - *Cache pointer*
  - *Get Free Page (GFP) flags*
- *kmem_cache_free*: Free an object and put it back in the cache.
  - *Cache pointer*
  - *Object pointer*: A pointer to the object that must be released.

Figure 2.2 shows a simplified workflow of the kernel object allocation through the slab allocator and the indirect communication between the kernel module and the page allocator sub-system.

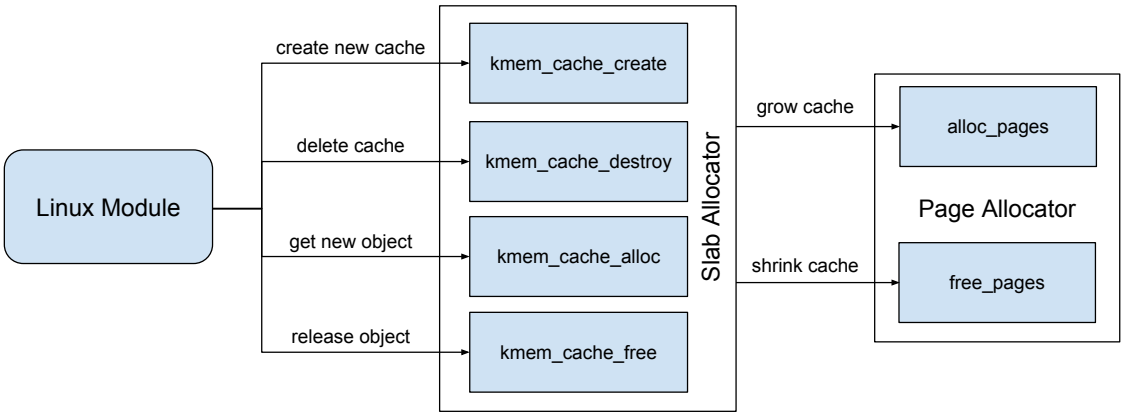

Figure 2.2: Slab Allocator Workflow

In Figure 2.3 the general memory layout of the slab allocators is illustrated. However, the internal implementation of the slab allocators might be different. For instance, the

SLOB allocator only uses a simple list for managing the free objects (K&R style) and only *emulates* the *SLAB* memory layout to provide a unified interface.

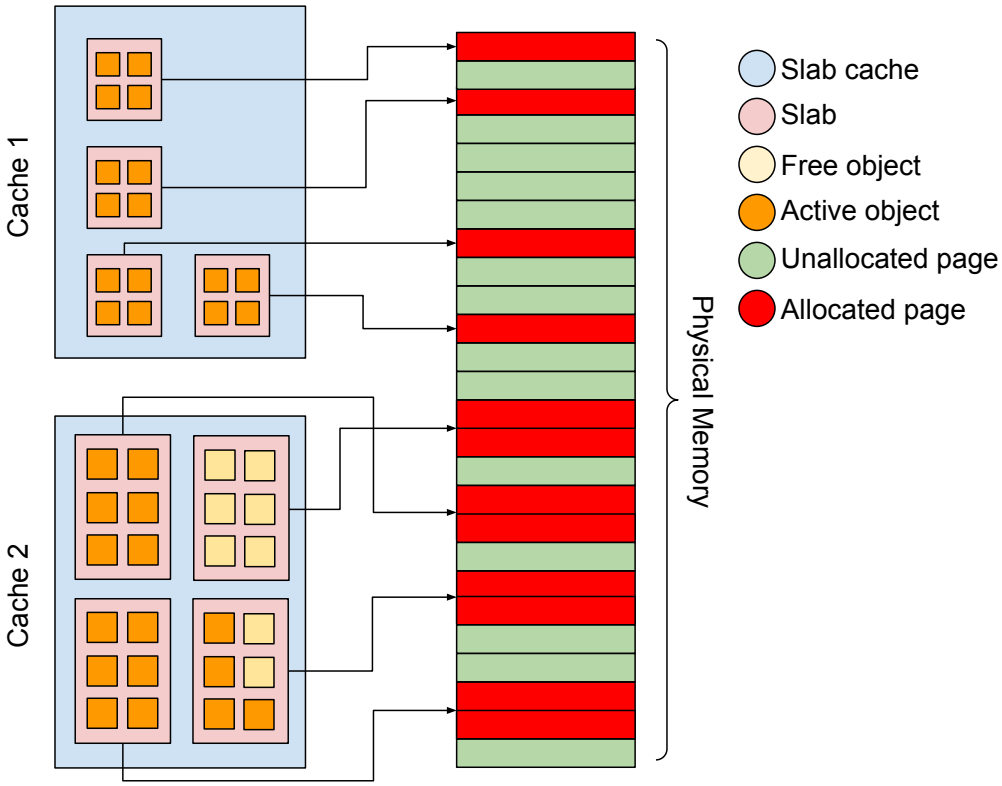

Figure 2.3: Slab Allocation Overview

As an example, let's assume that a kernel component such as network needs to allocate often and release an object of type `request_sock` for handling socket connection requests. Therefore, it makes a request to the slab allocator through the `kmem_cache_create` interface to create a cache of type `request_sock` struct so that it can satisfy subsequent memory allocations (and releases) on behalf of the network component. Based on the size of the struct, the slab allocator calculates the number of memory pages required for storing each slab cache (power of 2) and the number of objects that can be stored on each slab. Then, it returns a pointer of type `kmem_cache` to the network component as a reference to the created cache. At the time of creating a new cache, the slab allocator generates a number of slabs and populates them with the allocated and initialized objects (free objects). When the network component needs to create a new object of type `request_sock`, it makes a

request to the slab allocator through the `kmem_cache_alloc` function with the pointer (of type `kmem_cache`) to the cache. If the cache has a free object, it immediately returns the object that we call *fast path*. However, if all objects within the cache slabs are already in use (active), the slab allocator grows the cache by making a request to the *Page Allocator* through `alloc_pages` to get free pages. After receiving free pages from the page allocator, the slab allocator creates a one or more slabs (in the free physical pages) and populates them with the new allocated and initialized objects (*slow path*). On the other hand, at the time of releasing the active object, the Linux network component calls `kmem_cache_free` with the cache and object pointers as the parameters. The slab allocator marks the object as free and keeps the object in the cache for the subsequent requests (fast path). The free *slow path* will be taken if all the objects within a slab are free; the memory pages of that particular slab will be eligible for return to the free list of the free physical pages that is managed by the page allocator.

The Linux kernel also allows the `kmalloc` and `kfree` interfaces to retain compatibility with former modules that only use these functions for memory allocations. However, they no longer directly use the page allocator for memory allocations anymore. Instead, they are integrated into the slab allocator sub-system by providing a set of fixed size generic caches (kernel asks the slab allocator to create these generic caches on initialization). Moreover, these functions can be used for the objects that do not use a fixed structure (e.g., integer arrays, strings, etc.). These generic caches are displayed in `/proc/slabinfo` under the name of `kmalloc-<size>` and `kamlloc-dma-<size>` for `GFP_NORMAL` and `GFP_DMA` memory zones, respectively. The sizes of these caches vary from 8 to 8192 (8, 16, 32, 64..., 8192). Hence, the `kmalloc` interface satisfies the request of the memory allocation by obtaining a free object from a cache that fits the requested object size.

2.4.2 SLOB

SLOB(Simple List Of Blocks) is one of the slab allocators implementation that is created based on the conventional K&R heap allocator (The original kmalloc allocator in Linux before replacing it with Slab sub-system). It simply keeps track of the free objects in a doubly-linked list. SLOB satisfies the request of allocating a new kernel object by traversing the list of empty objects and finding the first block of sufficient size. In case of failure, it makes a request to the page allocator to grow the heap size. The SLOB allocator only emulates the SLAB layer insofar as it keeps everything in one list. The SLOB allocator also suffers greatly from internal fragmentation as well as other conventional K&R allocators. In 2008, SLOB sought to overcome this limitation by establishing a patch that replaced a single list with three lists of different sizes: small, medium, and large. The small source

code size of the SLOB in conjunction with the small memory footprint for managing the objects make this slab allocator a good choice for the embedded devices that have memory limitations.

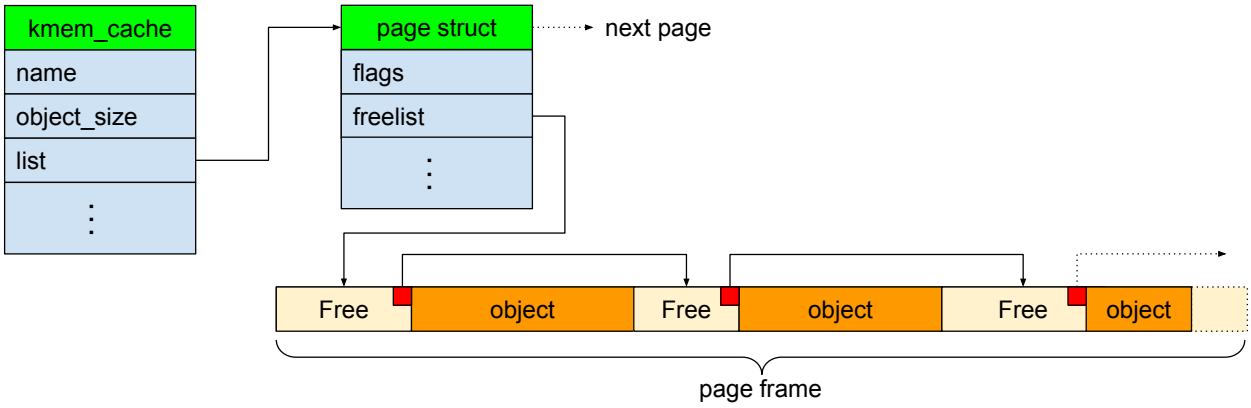

Figure 2.4: SLOB Memory Layout [32]

2.4.3 SLAB

SLAB is the name of the first slab allocator that was integrated into the Linux kernel. This implementation was the default Linux kernel slab allocator before SLUB. Although SLAB is well-known for its design philosophy and optimized CPU cache utilization, it wastes an enormous amount of memory for storing queues in multi-levels. SLAB uses a technique called *cache coloring* in which the initial offsets for the allocated objects within the slabs are different [32]. Since the slabs begin on page boundaries, the chance of mapping the slabs objects into the same CPU L1 cache is higher. Therefore, using this technique in the SLAB implementation prevents the possibility of cache *false sharing* [44] in the CPU L1 cache. The basic idea of the slabs cache coloring is shown in Figure 2.5.

The SLAB implementation has a complex data structure for managing free objects within the slab caches. It maintains per-CPU and per-node (NUMA node) queues to accelerate access to free objects. In SMP systems, the per-CPU queue (`array_cache`) that is available in the `kmem_cache` data structure provides a LIFO[1] queue of the free objects for each CPU. As an example, an object that is released on the "CPU 1" will be re-used (if possible) on the same CPU rather than other CPUs. The per-CPU queue works in an

[1]Last In First Out

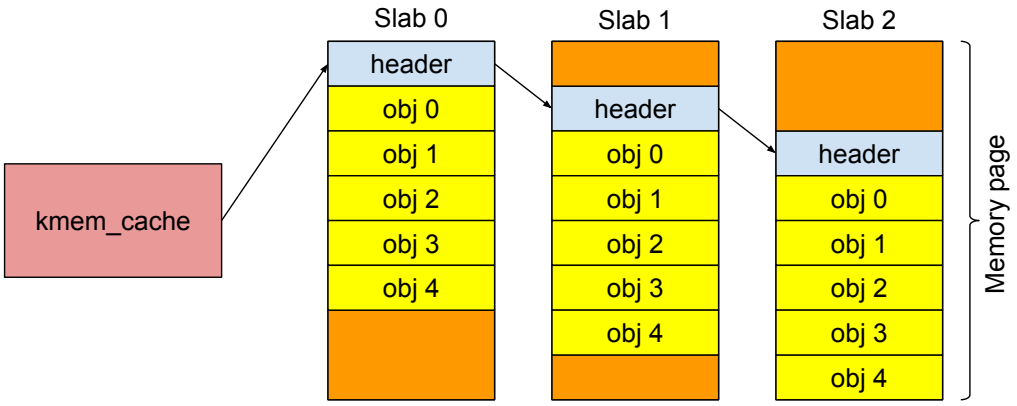

Figure 2.5: Slab Cache Coloring

LIFO ordering because it uses the cache warmed free objects to make the best use of the L1 CPU cache. Also, the per node data structure uses three lists of slabs (full, partial, free) to maintain slabs through the `page struct`. In the SLAB implementation, the free objects metadata (indices of the free objects) is stored at the beginning of each slab that imposes per-slab memory overhead. In Figure 2.6 we describe a simplified memory layout of the SLAB implementation.

When a kernel module asks for a free object from the cache, the SLAB allocator attempts to return a free object (if it has one) from the per-CPU free list (*fast path*). If the per-CPU free list runs out of the free objects, it makes a call to the `cache_alloc_refill` function to re-fill the per-CPU queue with the fresh free objects from either the `free_list` or the `partial_list` of the cache node (*slow path 1*). In the event that all free objects in cache node were consumed, the SLAB allocator will be forced to allocate the new memory pages using the page allocator, which proves more time-consuming (*slow path 2*).

2.4.4 SLUB

The SLUB is a *Unqueued* slab allocator that introduced in Linux kernel version 2.6.22 (in 2007) and became the default kernel slab allocator on many Linux distributions [13]. Although SLUB's memory layout is fairly similar to SLAB, it has a completely different philosophy with regard to implementing slab allocator techniques. In Figure 2.7 the memory layout of the SLUB slab allocator is shown.

We summarize the difference between SLUB and SLAB slab allocators in the list below [17, 31, 32].

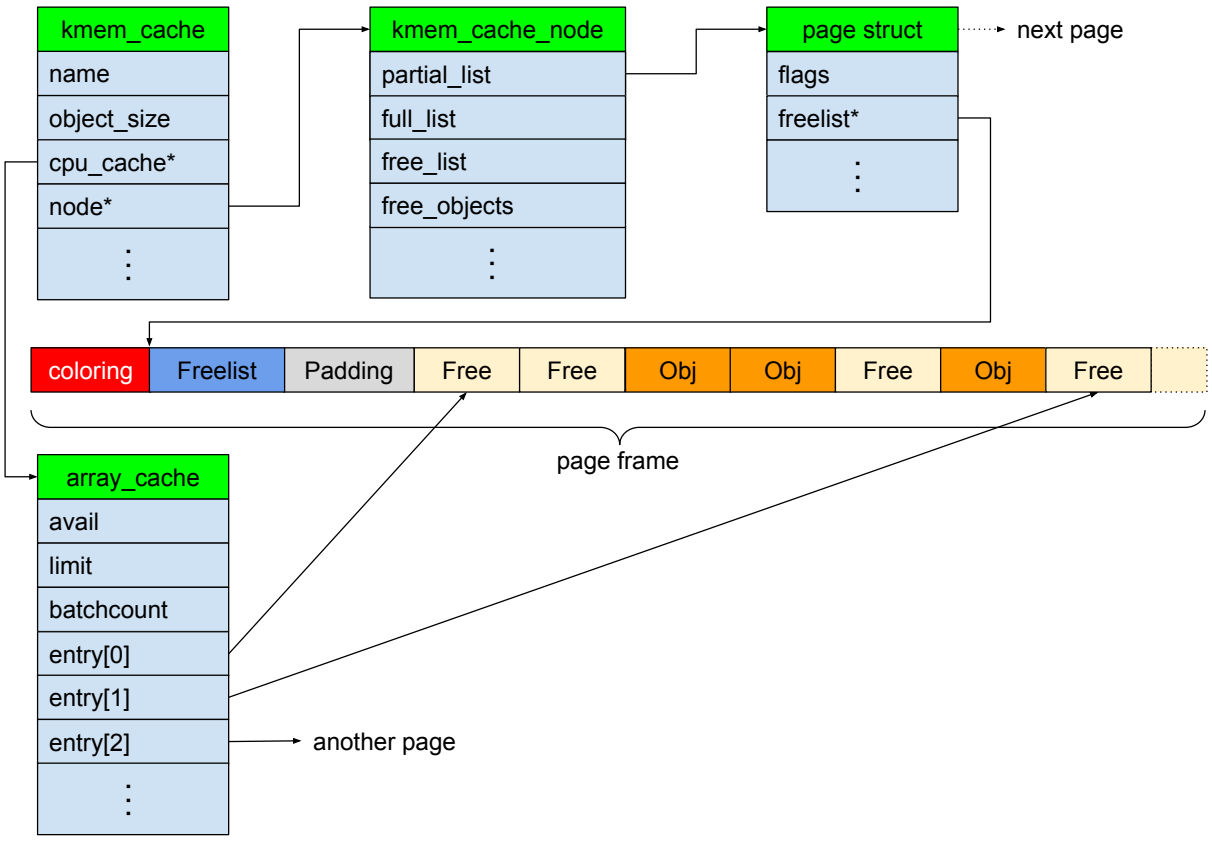

Figure 2.6: SLAB Memory Layout [32]

1. SLUB eliminates the need for per CPU and per node queues. Instead, it only retains a per CPU pointer for the first free object on the page.

2. Unlike SLAB, SLUB uses partial slabs in a per-CPU structure to improve CPU locality

3. SLUB *fast path* uses per-CPU data and `this_cpu` operations[14] to eliminate the need for disabling interrupts and taking mutex lock. (SLAB does not have this feature)

4. The per slab memory footprint is reduced in SLUB implementation due to the relocation of *freelist metadata* into the free objects.

5. SLUB supports debugging and defragmentation on multiple levels.

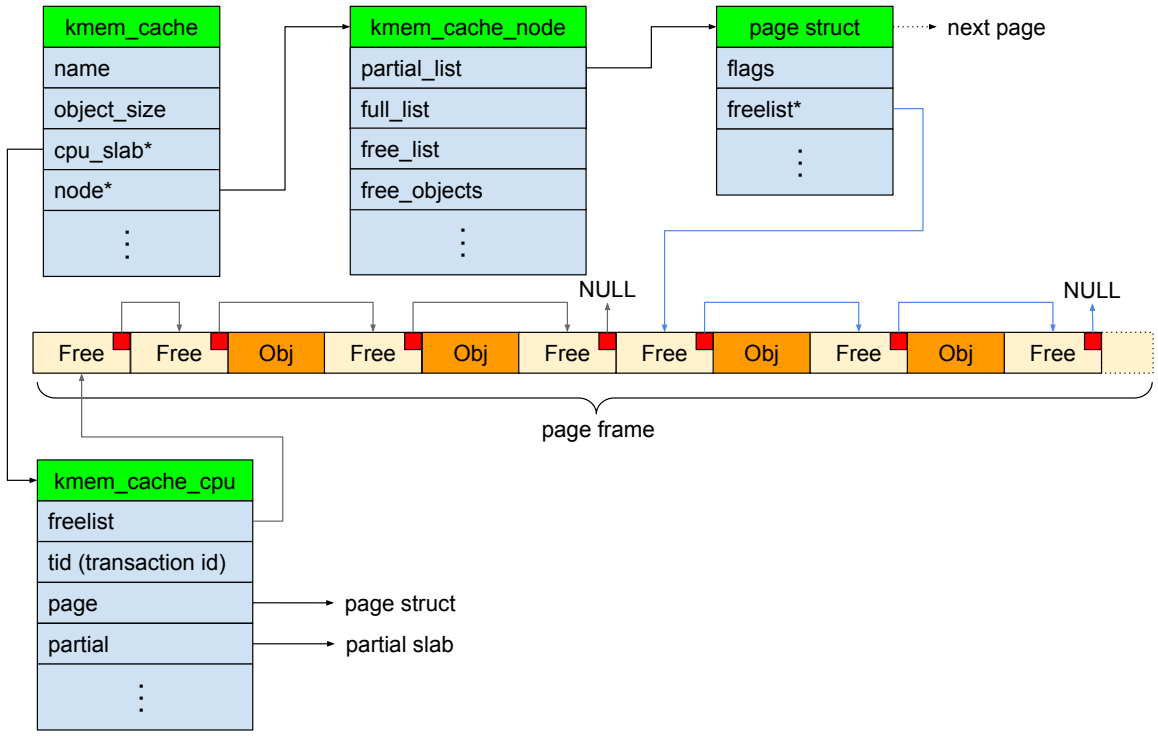

Figure 2.7: SLUB Memory Layout [32]

6. The cache aliasing feature in SLUB implementation reduces the memory overhead up to 50% by only unifying the caches of the same size.

7. Unlike the SLAB allocator, SLUB does not use cache coloring technique. However, SLUB reduces the cache line size to improve hardware cache performance.

2.4.5 Monitoring Slab Alocators

As we mentioned in Section 2.3, the `/proc/slabinfo` file provides the slab allocator statistics at run time. However, a tool named `slabinfo` that can be compiled from the Linux source code gives more information about the current status of the slab allocator (e.g., cache aliasing, fragmentation and debugging). These values are also available in the `/sys/kernel/slab` directory. Moreover, for real-time monitoring of slab allocator statistics, we can use a top-like tool called `slabtop`. The following shows a partial snapshot of

the `slabtop` output.

```
Active / Total Objects (% used)    : 422887 / 490308 (86.2%)
Active / Total Slabs (% used)      : 15169 / 15169 (100.0%)
Active / Total Caches (% used)     : 78 / 102 (76.5%)
Active / Total Size (% used)       : 100944.82K / 126741.17K (79.6%)
Minimum / Average / Maximum Object : 0.01K / 0.26K / 16.44K

 OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
79488  75681  95%    0.06K   1242       64      4968K kmalloc-64
67342  65050  96%    0.18K   3061       22     12244K vm_area_struct
51891  38420  74%    0.19K   2471       21      9884K dentry
48468  20864  43%    0.57K   1731       28     27696K radix_tree_node
31926  30514  95%    0.08K    626       51      2504K anon_vma
28704  28352  98%    0.10K    736       39      2944K buffer_head
26248  25582  97%    0.12K    772       34      3088K kernfs_node_cache
```

If the kernel is compiled with the SLUB allocator, the *slabinfo* tool will also enable us to modify the tunable options within each cache and report the status of available caches.

2.5 Binning Linux Slabs

In Section 2.2, we briefly explain the different forms of binning that can happen across the Linux kernel. Since the slab allocator became the core component to interact with the main memory for allocating small objects in the kernel, we decided to investigate the binning effect of the slab allocator on the user-space programs that interact with the kernel through the system calls. To be more precise, we perform some experiments on a wide range of system calls to see which of them are subjected to the binning effect imposed by the slab allocator. Therefore, we can conclude that user-space programs that a specific system call are prone to greater execution time variability compared to programs that do not use that system call. Figure 2.8 shows user-space programs interaction with slab allocators through the Linux kernel system calls.

2.5.1 Experiments

We start our experiments with a simple test that calls the `mmap` system call 1000 times and measures the execution time of each iteration of the system call. Since `mmap` only *reserves*

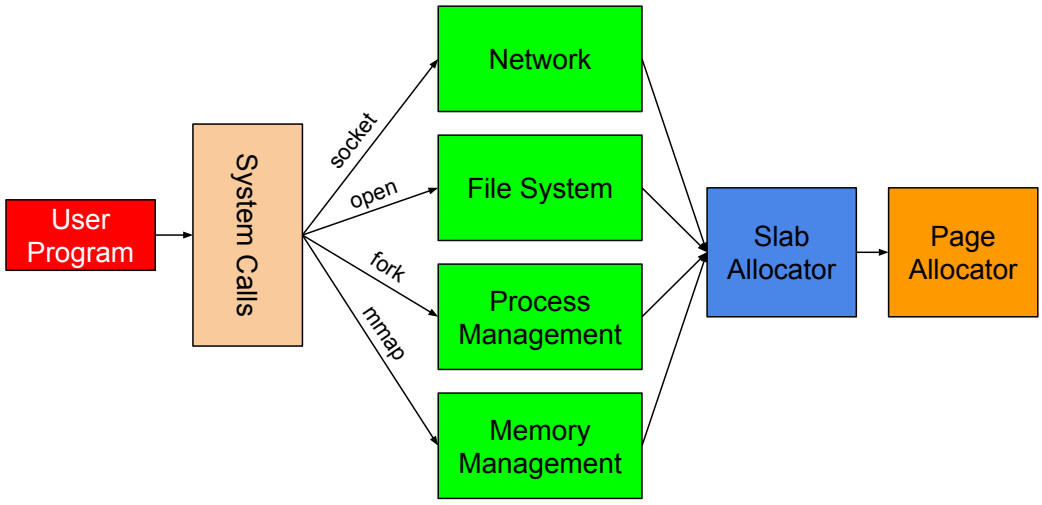

Figure 2.8: Indirect interaction between user-space programs and slab allocator

the address from the process address space, we do not expect it to actually allocate any page from the memory. However, for keeping track of the memory mappings, the kernel stores an object of type `vm_area_struct` for each call to the `mmap` function. Kernel uses the slab allocator to allocate free objects of type `vm_area_struct`. In this experiment, we use two different calling patterns to the `mmap` system call to investigate the possible slab allocator binning. In the first calling pattern, we make a call to the `mmap` system call 1000 times without using `unmmap` to *delete* the mapped address (Binning). On the other hand, in the second program, to prevent the binning effect, we call `unmmap` after each `mmap` system call to delete the mapping (from user-space) and return the VMA object (in the kernel) to the object cache (No Binning). The results are shown in Figure 2.9.

The binning effect appears in the `Binning` program as we observe the execution time outliers (marked by cross) that happen in exactly a fixed distance from each other after 93 calls to the `mmap` system call. The outliers we observe in this experiments occur *every 60* calls to the `mmap` function. Since the Linux kernel of the machine that runs the experiment is compiled with the SLAB, we need to examine the bin(s) that trigger the slow path in that particular slab allocator. As we explained earlier in Section 2.4.3, the first slow path can be taken when the per-CPU "freelist" runs out of free objects and the second slow path must be taken when all the free objects in per node slabs are consumed. In this case, the magic number 60 is equal to the `batchcount` tunable field of the `array_cache` per-CPU structure. Also, the value of the `limit` field of the same structure (that is also

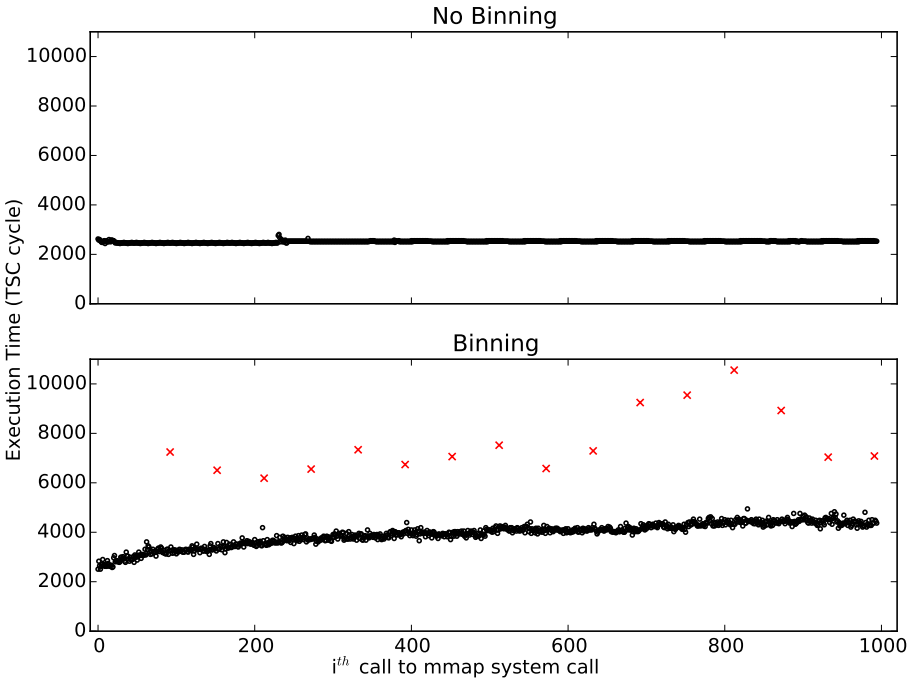

Figure 2.9: Bin Vs. Nobin

tunable) is equal to 120 that can explain why the first binning happens at the 94^{th} call. Therefore, we can conclude that the other 25 per-CPU free objects of type `vm_area_struct` were consumed before the execution of the loop.

We also perform the same *Binning* and *No Binning* experiment on other system calls that have an interface similar to `mmap/munmap`. For instance, `open/close` was one of the candidates that we investigated by conducting an experiment on two programs with different calling patterns to the `open` and `close` system calls. The results are shown in Figure 2.10.

Surprisingly, in the *No Binning* program, we can still see the patterned outliers that indicate the slab allocator footprint. By tracing the `open` and `close` system calls in kernel source code, we discovered that the `close` system call does not synchronously clear the memory footprint of the opened file. Instead, it only removes the link to the process file descriptor table and postpones the call to the `fput` functions (which releases the kernel object) by adding the task to the kernel *workqueue* through the `schedule_delayed_work` function. Therefore, the call to the `kmem_cache_free` function for releasing the kernel object is made in an *asynchronous* fashion that results in the slab allocator binning effect.

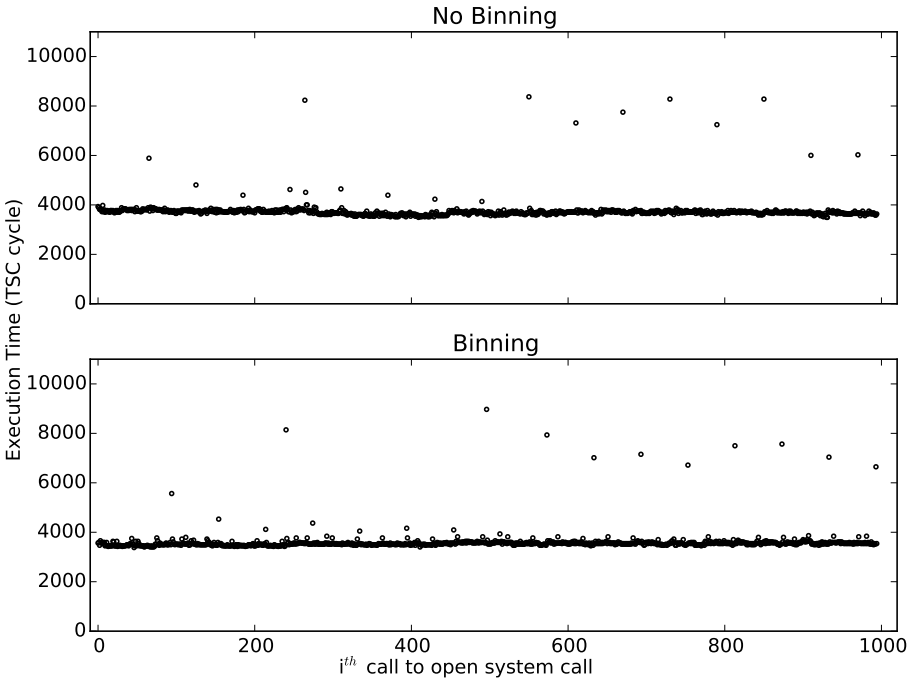

Figure 2.10: Binning vs. No Binning - `open` system call

We also conduct a one-way ANOVA study on the results of `mmap` and `open` system call to see whether or not the calling pattern has a significant effect on the execution time variability or not. Therefore, we can write the null hypothesis as follows:

Null Hypothesis: There is no significant difference between the mean (mean of variations) of the two groups. (Binning and No Binning)

The ANOVA summary of `mmap` and `open` experiments are shown in Tables 2.1 and 2.2, respectively.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Binning Effect	1	614.18	614.18	3.9302	0.08273
Residuals	8	1250.17	156.27		

Table 2.1: `mmap` ANOVA Analysis

The ANOVA results from the `open` system call experiment indicate that we cannot reject the null hypothesis at the level of significance 0.05. In other words, there is no

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Binning Effect	1	171108	171108	2684	2.135e-11
Residuals	8	510	64		

Table 2.2: `open` ANOVA Analysis

significant difference between the "Binning" and "No Binning" groups. On the other hand, in the `mmap` experiment, the p value of 2.135e-11 shows that call patterns significantly impact execution time variation.

In the second part of our experiments on Linux kernel binning effect we study the binning effect of the slab allocator on more than 40 widely used system calls. Our approach commences with a definitions of a new metric called slab metric that represents the interaction between a particular system call and the slab allocator. For example, by performing static or dynamic analyses on the `getpid` system call, we can determine any calls to the slab allocator interfaces. Therefore, we can assign a value of zero to the *slab metric* of this function as the `getpid` function does not use the slab allocator for memory allocation. Then, we continue our experiments by conducting a dynamic and static analysis on each system call to find the slab metric value for each approach. Finally, by running the benchmark programs and measuring the execution time variation of each system call, we examine the possible correlation between the *slab metric* and *execution time variation*.

For dynamic analysis of the system calls we write one benchmark program for each system call that calls the method with the specified arguments. Also, we use *ftrace* Linux function tracing tool to dynamically monitor the execution of these benchmark programs. Since the slab allocator has only a few functions for allocating or releasing the kernel objects, we only configured the *ftrace* tool to monitor the calls to the following slab allocator functions.

- `kmem_cache_alloc`
- `kmem_cache_free`
- `kmalloc`
- `kfree`

The only limitation of this approach that is also applied to other dynamic analysis techniques is that we could only analyze one execution path of the function (depends on the

passed arguments). Therefore, our dynamic analysis results are limited to only one form of calling the system call functions. Moreover, we do not consider the requests for allocating temporary objects (memory) from the slab allocator in our slab metric assignment. As an example, the `open` system call allocates a memory object of size `PATH_MAX` (4094) to temporary store the path of the requested file to open. However, it releases the object to the cache after opening the file. The snippet below shows the calls to the slab allocator interfaces from the `open` system call. The results are captured via *ftrace* function profiler.

```
kmem_cache_alloc: (getname_flags+0x37) bytes_req=4096 gfp_flags=GFP_KERNEL
kmem_cache_alloc: (get_empty_filp+0x5c) bytes_req=256 gfp_flags=GFP_KERNEL|GFP_ZERO
kmem_cache_free:  (putname+0x5b)
```

Therefore, the number we assign as the dynamic slab metric for the `open` system call is one (The first allocation is canceled out by the last call to free). The other factors that we do not consider (for the sake of simplicity) in dynamic slab metric assignment are the size of the requested object and the memory allocation GFP flags. In fact, these factors might have a significant contribution to the execution time of the slow paths that eventually could affect the total variation.

In static-based analysis approach we only use the source code of the Linux kernel for assigning a number to each Linux system call that represents the size of interaction with slab allocator memory management sub-system. We achieve this by extracting the callgraph of each system call function from the Linux source code using a call graph generation utility for C and C++ named CodeViz[4]. In contrast, the static analysis approach provides a more comprehensive view of possible execution paths by examining the complete source code of the kernel. However, on disadvantage of this method emerges out of the fact that the size of the callgraph grows exponentially by tracing the paths that never get executed (e.g., debug enabled paths). Therefore, we pruned the call graph tree by removing paths that never reach the slab allocator functions. We show a pruned callgraph of the `mmap` system call obtained from the Linux kernel (version 3.12.12) in Figure 2.11.

The static approach to the slab metric assignment is not as easy as the dynamic approach since knowledge of possible execution paths that will eventually interact with the slab allocator sub-system are unknown in the preliminary level of static analysis. We simply use the number of distinct paths to the slab allocator interfaces from the root symbol of the system call tree. Therefore, our naive method in assigning the slab metric to each system call might contain *false positive* error as we assumed an equal chance of taking all the paths (reporting high slab metric while the system function does not interact with the slab allocator). Moreover, since we could not generate the call graph for the function calls in

the kernel that made by reference, our results are also subjected to the *false negative* (e.g., reporting slab metric of zero while the system call uses the slab allocator sub-system). In conclusion, the slab metric in static approach is not as accurate as the dynamic approach.

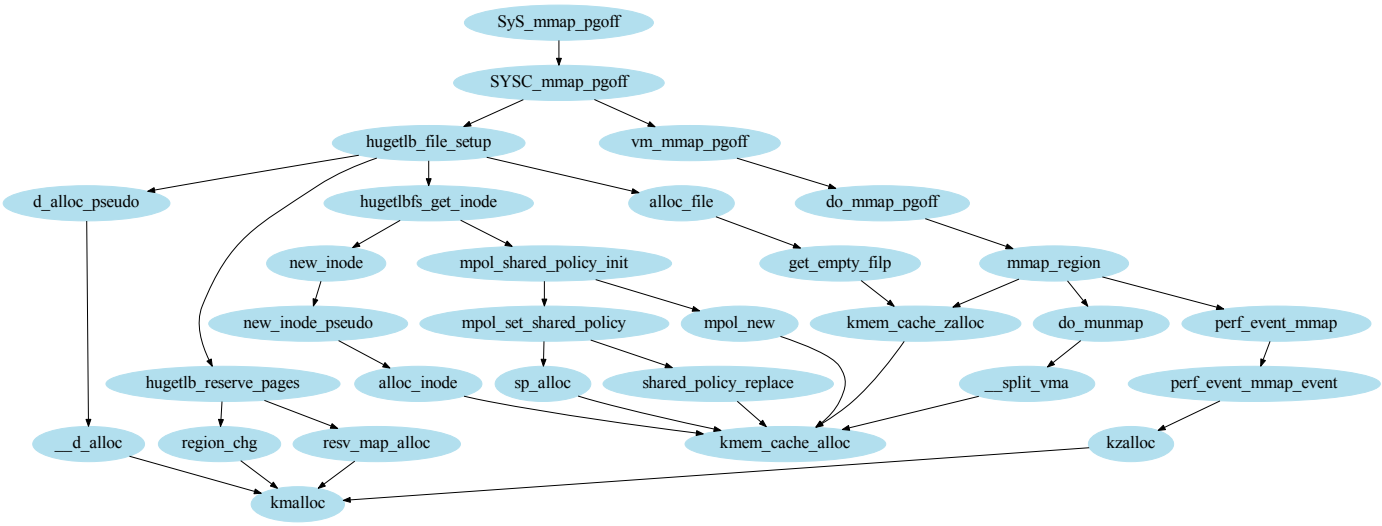

Figure 2.11: Pruned callgraph of Linux `mmap` system call

2.5.2 Microbenchmarks Program

To demonstrate the impact of binning that is imposed by the slab allocator, we draw inspiration from the *Libmicro* microbenchmarks architecture in our microbenchmark design. However, we remove unnecessary process and thread synchronization functions to reduce the factors that can affect the variation. Therefore, we implement a simplified version of the *Libmicro* that runs only on a single process and a single thread. The snippet below shows a pseudo code of the Libmicro-inspired structure of our microbenchmarks.

```
1 init_benchmark()
2   FOR counter = 1 to 1000
3     start_batch()
4
5     start = rdtsc()
6     benchmark()
7     end = rdtsc()
8
9     times[counter] = end - start
```

```
10
11      end_batch()
12    ENDFOR
13  finish_benchmark()
14
15  print times
```

Snippet 2.1: Benchmark program pseudo code

Since the accuracy of recording the elapsed time of executing each iteration of the system call is a critical factor in our experiment, we selected hardware TSC as a counter that could deliver the highest accurate result. The TSC is a 64-bit hardware register that is available on all x86 processors. Due to known possible flaws in the TSC, we are encouraged to first, select the *user-space* CPU governer and set a fixed frequency to prevent any CPU frequency scaling (e.g., power-saving mode), and second, run the program on only one CPU core as there is no guarantee that the TSC register of multiple CPU cores on a single processor socket will be synchronized [7]. In order to read the value of the TSC register with the lowest possible overhead, the x86 ISA offers `rdtsc` and `rdtscp` instructions. The `rdtscp` instruction that is only available on the recent CPUs, prevents instruction reordering (out-of-order execution) around the call to this instruction [15]. We use `rdtscp` instruction in our implementation of the `rdtsc` function.

```
1  __inline__ unsigned long long
2  rdtsc(void)
3  {
4    unsigned long long ret;
5    asm volatile ( "RDTSCP" : "=A"(ret) );
6    return ret;
7  }
```

Snippet 2.2: `rdtsc` function

The `benchmark` function is implemented individually for each target system call. Also, the `rdtsc` and `benchmark` functions are defined as "inline" functions to avoid function call overhead.

2.5.3 Testing Environment

Since in this experiment we are *only* interested in the effect of the slab allocator Linux sub-system on the program execution time variation, we try to eliminate *all* other sources (hardware or software) that may affect our experiment. On the hardware side, we have

a number of elements (e.g., CPU cache and CPU frequency scaling) that can potentially mask out our desired factor. However, on the software side, we are facing a variety of factors such as process context switch, software or hardware interrupts and kernel locking in the SMP environment. We use the Datamill benchmarking infrastructure to ensure that our experiment will be run in a clean and controlled environment. Also, since the TSC counter is only available on x86 machines, we are limited to running our experiment on these machines. In preemptive kernels, the scheduler is permitted to perform the context switch at any time, even in in the midst of executing a system call. Therefore, performing a context switch during system call execution could result in outliers that are even greater than the slab allocator slow path outliers by several orders of magnitude. In these special cases, we removed the data points (outliers) affected by context switch from the final results. To make this process automated, we count the total number(n) of context switch that occurs during the execution of the program's loop through the `getrusage` system call and remove n maximums data points from the raw results.

2.5.4 Results

In this section we present the measurement results of our experiments and the slab metrics we computed based on dynamic and static analysis approaches. Since the measured mean of the execution times (system calls) is not equal, we also report the "RSD" or "Coefficient of Variation (CV)" that shows the variability in relation to the mean of the population. RSD can be calculated using the formula below.

$$RSD = \frac{\sigma(standard\ deviation)}{\mu(mean)} \tag{2.1}$$

The results are shown in the Table 2.5.4. The *Static Metric* and *Dynamic Metric* are referred to the *Slab Metrics* that are calculated based on static and dynamic analyses, respectively.

System Call	Static Metric	Dynamic Metric	SD(cycles)	RSD
accept	17	2	775.31	12.84
bind	8	3	5,401.91	14.34
brk	23	0	69.64	5.70
chdir	0	0	148.67	4.93
close	1	0	78.76	5.96

System Call	Static Metric	Dynamic Metric	SD(cycles)	RSD
dup	6	0	190.37	28.96
getcwd	1	0	10.10	1.05
getpid	0	0	4.78	5.14
getrusage	1	0	12.63	0.96
getsockname	6	0	258.47	11.03
gettimeofday	0	0	11.49	6.68
listen	0	0	253.16	9.49
lseek	0	0	27.58	7.31
mkfifo	0	0	53.15	3.12
mknod	0	0	25.82	1.15
mlock	11	0	198.17	5.70
mmap	14	1	663.66	16.67
msgget	0	0	10.50	2.57
msgrcv	3	1	311.53	32.15
msgsnd	6	1	228.94	27.77
munlock	8	0	231.28	6.05
munmap	7	1	381.00	15.25
open	13	1	411.22	11.45
pipe	42	6	2,209.85	36.91
read	7	0	18.17	1.65
readv	11	0	10.40	0.96
realpath	0	0	14.75	1.35
rename	57	3	9,176.14	31.77
semctl	10	0	44.15	4.60
semget	0	1	483.35	25.62
semop	0	0	10.85	1.31
sethostname	0	0	7.75	0.87
setsocketopt	0	0	11.41	2.02
sigaction	0	0	11.72	2.07
signal	0	0	12.41	0.30
signalfd	8	3	751.09	38.43

System Call	Static Metric	Dynamic Metric	SD(cycles)	RSD
sigprocmask	0	0	9.74	2.09
socket	18	5	1,733.03	42.42
unlink	30	1	754.17	4.16
write	7	0	13.36	2.44

Table 2.4: Slab Metrics vs. RSD

Also, Figure 2.12 shows the correlation plot of the slab metrics and the RSD. (The dynamic and static metrics are scaled to 10)

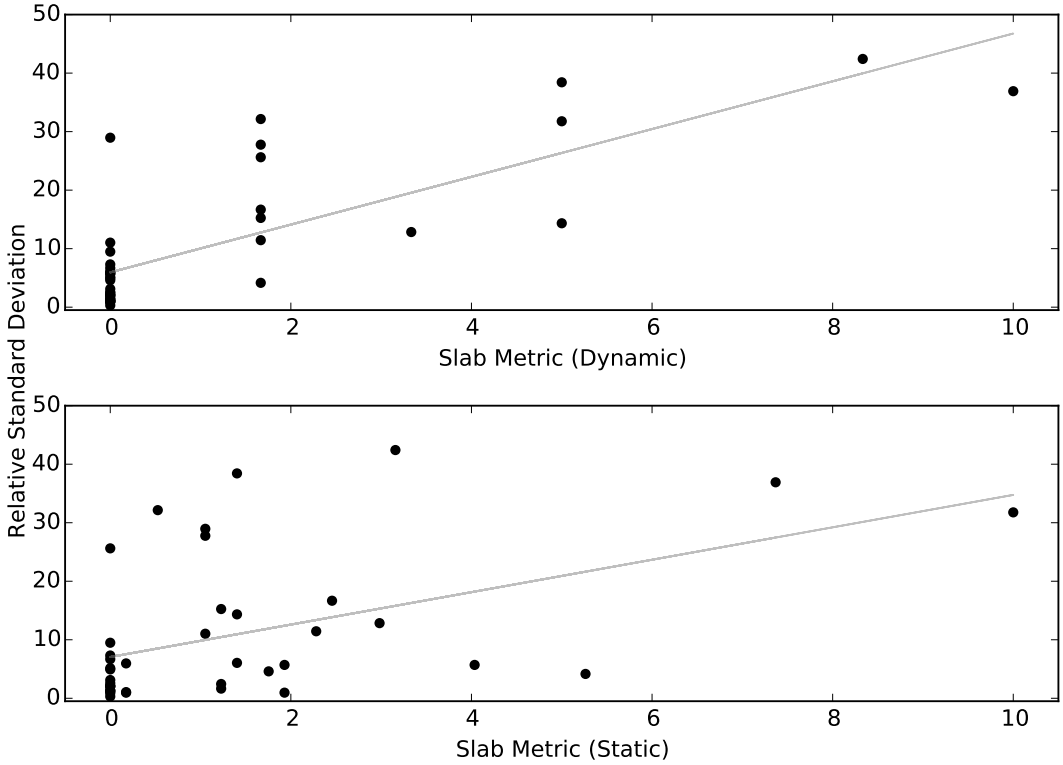

Figure 2.12: Slab Metric vs. Variability (Correlation)

The correlation coefficient (r) of linear regression for dynamic and static analysis are **0.784** and **0.431**, respectively.

Chapter 3

Conclusion and Future Work

3.1 Thesis Summary

The act of optimizing a software program to run as fast as possible on a given hardware is not a trivial task. The primary step in tackling an optimization task is to carefully analyze the performance of the program by conducting a set of measurement-based experiments. However, recent architectural changes to software and hardware platforms such as NUMA and SMP comlicates this this task. For instance, the TSC hardware counter, which can be used to measure the execution cycles of the programs, is not synchronized among the CPU cores. Consequently, incorrect results may arise when the program execution migrates from one core to another. Therefore, selecting a suitable approach and tool for performing software performance analysis is necessary.

In Chapter 1, we investigate a variety of measurement-based performance analysis methods with emphasis on hardware performance counters that are widely used in both industry and academy. We explain the methods of accessing hardware counters on different architectures and broadly explore the functionality of the perf_event as the only long-lived PMU interface that is merged to the Linux kernel mainline. To evaluate the Linux Perf tool in terms of accuracy, determinism and overhead, we conduct a set of experiments that compare the Perf results with other methods such as dynamic binary instrumentation and hardware traces. On the subject of determinism in counting mode, our experiments show that the only hardware event count that is deterministic is "taken branch" on the "Intel Pentium 4" machine. However, the same event on the same machine seems to have 47% error in reporting the total number of counted events when we compared with the Pin results. Regarding overhead, we observe up to 5.92% relative error on the Intel Pentium 4

machine in running `perf record` with the default sampling options. Moreover, we realized that the type of hardware event does not have any impact on the overhead as long as we do not change the sampling rate. Finally, regardless of the inaccuracy and overhead that we observed on the "Intel Pentium 4" that is related to the implementation of the PMU in hardware level [43], we found the Linux Perf tool and its underlying perf_event sub-system an accurate, low-overhead and easy-to-use method for performing measurement-based performance analysis.

In Chapter 2 we study the effect of memory binning caused by the slab allocator memory management technique on execution time variability of the Linux system calls. We define a metric for each system call based on the number of requests to the slab allocator and try to find a correlation between the execution time variation of the system calls and the computed metric. In our experiments, we attempted to eliminate other possible sources of timing variation in Linux kernel by running the benchmark programs in a clean and controlled environment in both software and hardware aspects. The results showed a stronger correlation coefficient for slab metric that we obtained from the dynamic analysis approach than the static method.

3.2 Future Work

Our study in Chapter 1 was limited to only three hardware events, named *CPU cycles*, *retired instructions* and *taken branches*. However, in the modern processors, there are many hardware events such as cache and memory related events that require investigation. We would also like to explore the overhead and accuracy of the *PMU* and test the Linux perf_event interface on other architectures such as MIPS, ARM, and Power. Moreover, as we mentioned in Section 1.8, the other applications of the hardware traces for conducting performance analysis in hard real-time systems will remain for future works.

The basic idea of the binning effect in the Linux kernel that we partially explore in Chapter 2 was a unique research topic that remains unexplored at present. As we point out in Section 2.2, the Linux kernel uses a variety types of binning that could potentially affect the timing behavior of the programs that are running on the system. In this thesis, we only explore a particular kind of memory binning that is imposed by a sub-system of the kernel memory management called the slab allocator. Also, during our exploration of Linux standard libraries such as *libc* we found some functions (e.g., `malloc` and `free`) that use a binning mechanism in calling the `brk` system call for growing and shrinking the heap size. Therefore, the binning effect is not only limited to the kernel level, but also includes the low-level libraries (i.e., `malloc`) that need to be further investigated in future studies.

References

[1] Arm coresight architecture specification. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0029d/IHI0029D_coresight_architecture_spec_v2_0.pdf.

[2] Arm information center. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka4237.html.

[3] Arm information center. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka4237.html.

[4] Codeviz: A callgraph visualiser. http://www.csn.ul.ie/~mel/projects/codeviz/.

[5] Cortex-a15 technical reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0438d/BIIBDHAF.html.

[6] Hardware performance counter. https://en.wikipedia.org/wiki/Hardware_performance_counter.

[7] Intel 64 and ia-32 architectures software developers manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf.

[8] Model-specific register. https://en.wikipedia.org/wiki/Model-specific_register.

[9] Open benchmarking platform — datamill. https://datamill.uwaterloo.ca.

[10] Oprofile documentation. http://oprofile.sourceforge.net/docs/.

[11] Papic:papi avail.1 - papidocs. http://icl.cs.utk.edu/projects/papi/wiki/PAPIC:papi_avail.1.

[12] perf_event_open - linux manual page. http://man7.org/linux/man-pages/man2/perf_event_open.2.html.

[13] Slub: The unqueued slab allocator v6. http://lwn.net/Articles/229096/.

[14] This cpu operations. https://www.kernel.org/doc/Documentation/this_cpu_ops.txt.

[15] Using the rdtsc instruction for performance monitoring. https://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf.

[16] Xilinx zynq-7000 all programmable soc zc702 evaluation kit. http://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html.

[17] The slub allocator. https://lwn.net/Articles/229984/, April 2007.

[18] R. Berrendorf and H. Ziegler. Pcl the performance counter library: A common interface to access hardware performance counters on microprocessors. 1998.

[19] Jim Blakey. Overview of linux memory management concepts: Slabs. http://www.secretmango.com/jimb/Whitepapers/slabs/slab.html.

[20] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. *USENIX Summer*, pages 87–98, 1994.

[21] M. Jovic D. Zaparanuks and M. Hauswirth. Accuracy of performance counter measurements. *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, pages 23–32, April 2009.

[22] J. Demme and S. Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. *Proc. 38th IEEE/ACM International Symposium on Computer Architecture*, June 2011.

[23] L. DeRose. The hardware performance monitor toolkit. *Proc. 7th International Euro-Par Conference*, pages 122–132, 2001.

[24] S. Eranian. Perfmon2: a flexible performance monitoring interface for linux. *Ottawa Linux Symposium*, pages 269–288, July 2006.

[25] M. Goda and M. Warren. Linux performance counters pperf and libpperf. 1997.

[26] Brendan Gregg. Perf examples. http://www.brendangregg.com/perf.html.

[27] D. Heller. Rabbit: A performance counters library for intel/amd processors and linux. 2001.

[28] E. Hendriks. Perf 0.7. 1999.

[29] H. Hoyer. p5 performance counter msr driver. 1998.

[30] G. Hager J. Treibig and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. *Proceeding of the First International Workshop on Parallel Software Tools and Tool Infrastructures*, September 2010.

[31] Joonsoo Kim. How does the slub allocator work. http://events.linuxfoundation.org/images/stories/pdf/klf2012_kim.pdf, 2012.

[32] Christoph Lameter. Slab allocators in the linux kernel: Slab, slob, slub. http://events.linuxfoundation.org/sites/events/files/slides/slaballocators.pdf, October 2014.

[33] J. Levon. Oprofile.

[34] L. Salayandia M. Maxwell, P. Teller and S. Moore. Accuracy of performance monitoring hardware. *Proc. Los Alamos Computer Science Institute Symposium*, October 2002.

[35] Terje Mathisen. Pentium secrets. http://www.gamedev.net/page/resources/_/technical/general-programming/pentium-secrets-r213, July 1994.

[36] D. Mentre. Hardware counter per process support. 1997.

[37] M. Pettersson. The perfctr interface.

[38] Christine Deane Philip J. Mucci, Shirley Browne and George Ho. Papi: A portable interface to hardware performance counters. 1999.

[39] K. London S. Moore, D. Terpstra. papi deployment, evaluation and extensions. *Proc. of User Group Conference,*, June 2003.

[40] L. Salayandia. A study of the validity and utility of papi performance counter data. Master thesis, The University of Texas at El Paso, 2002.

[41] Dan Terpstra Vincent M. Weaver and Shirley Moore. Non-determinism and overcount on modern hardware performance counter implementations. *ISPASS*, 2013.

[42] Vincent M. Weaver. Linux perf event features and overhead. *FastPath Workshop*, 2013.

[43] Vincent M. Weaver. Self-monitoring overhead of the linux perf_event performance counter interface. *ISPASS*, 2015.

[44] Michael L. Scott William J. Bolosky. False sharing and its effect on shared memory performance, September 1993.