UNIVERSITÉ CÔTE D'AZUR

MASTER INFORMATIQUE

TER report

Hardware Performance Counters

Prepared by
Francois Flandin

Supervised by
Sid Touati

Master Informatique
at Université Côte d'Azur
Second semester of 2024-2025

Abstract

This report investigates the use of Hardware Performance Counters (HwPCs) to measure and analyze program performance, as well as to gain insight into their underlying implementation.
Two C programs—a matrix multiplication and a simple addition loop—were used to represent memory-bound and CPU-bound workloads, respectively.
These benchmarks were executed using two different methods: the Linux `perf` command-line tool and the `libpfm` C library. The results demonstrate that `libpfm` provided more accurate and flexible performance measurements, making it a more effective tool for identifying performance bottlenecks.

Contents

1 Introduction

The tutorship project "Hardware Performance Counters" explores how CPUs can mesure their performances using Performance Monitoring Units.
The CPU, when working, triggers some "events" that the Performance Monitoring Units can detect, count, and store.
This project is divided in several phases :

CPU Analysis: Explore CPU's Hardware Performance Counters capabilities.

Study: Explore how Hardware Performance Counters works in a CPU.

Benchmarks: Use performance counters to benchmark two programs : one CPU-bound and one Memory-Bound.

The project is a big focus on `x86` architecture for simplicity, other architectures like ARM does not work the same with performance counters. And the tests were run on two machines, one with an Intel CPU, and the other with an AMD CPU.
If you wish to explore the details of the work done for the project, the project's code is hosted on GitHub at https://github.com/omelette-bio/hardware-performance-counters. It includes tools for CPUID analysis, event listing, and benchmark automation.

2 State of the Art

High performance processors depend on Hardware Performance Counters (HwPCs) for gaining deep insights into microarchitectural behavior during program execution. These special-purpose registers, embedded inside the CPU, monitor important events like CPU cycles, retired instructions, cache misses, branch mispredictions, and memory subsystem activity. By taking such measurements, HwPCs allow developers and researchers to diagnose performance bottlenecks, optimize software, and even reveal security weaknesses.

The use of HwPCs is not consistent across processor vendors. Intel's Performance Monitoring Unit (PMU), for example, supports sophisticated features such as Precise Event-Based Sampling (PEBS), which reduces profiling overhead while gathering accurate instruction-level information. Additionally, Intel's Top-Down Microarchitecture Analysis (TMA) employs HwPCs to categorize pipeline inefficiencies, assisting developers in determining if stalls are the result of front-end bottlenecks, poor speculation, or memory latency. AMD's Performance Monitoring, on the other hand, features Instruction-Based Sampling (IBS), a novel feature that offers fine-grained instruction fetch and execution behavior. AMD also exposes NorthBridge counters which are quite useful to analyze interconnect contention and memory bandwidth.

While this project is targeted for x86 architectures, it is noteworthy that ARM processors also incorporate PMUs, but with fewer standard events and other access methods. HwPCs are accessible at various levels of abstraction. At the lowest level, x86 offers machine instructions such as RDPMC and WRMSR to read and set up performance counters directly. The majority of developers, however, interact with HwPCs using higher-level interfaces, such as the Linux kernel's perf event open system call or the perf command-line utility, which abstract aside architectural aspects. Libraries such as libpfm and PAPI (Performance API) also ease event encoding, whereas commercial tools such as Intel VTune and AMD uProf provide graphical interfaces for performance analysis.

The uses of HwPCs go well beyond conventional performance tuning. In security research, they have been instrumental in finding side-channel attacks, like Spectre and Meltdown, by revealing unusual cache access patterns. Energy efficiency is another evolving region of interest—most modern CPUs have RAPL (Running Average Power Limit) counters, which convert performance events into power usage, making computing greener. Even machine learning frameworks are helped by HwPCs; frameworks such as TensorFlow and PyTorch utilize them to profile kernel execution and tune deep learning workloads.

For all their usefulness, HwPCs are not without their difficulties. A significant limitation is the lack of physical counters—CPUs have just 4–8 general-purpose counters per core, so users will have to time-multiplex measurements or rate-limit certain events. Architectural diversity exacerbates the issue; event encodings and availability differ not just between Intel and AMD but also across microarchitectures (e.g., Skylake and Zen). Documentation gaps form another difficulty because some events are poorly documented or reserved for internal diagnostics. Finally, although HwPCs strive to achieve low overhead, certain profiling techniques (e.g., PEBS or IBS sampling) can produce measurement biases when overused.

Recent developments keep widening the scope of HwPCs. Intel's hybrid architectures (e.g.,Alder Lake) now have core-type specific counters for Performance (P) and Efficiency (E) cores reflecting the growing complexity of modern CPUs. Meanwhile, AI-driven tools like AMD uProf and Intel Advisor are starting to automate performance analysis, recommending optimizations based on HwPC data. Beyond conventional profiling, researchers are finding new applications for example, integrating HwPCs with eBPF for kernel behavior monitoring in real time or using them to harden systems against microarchitectural attacks.

In the future, the sector is moving towards more standardization. Efforts like the PMU Tools Project strive to standardize event semantics between vendors, so that cross-platform benchmarking more reliable. Energy-aware profiling is another possible avenue, as data centers need improved tools for a balance of speed and power conservation. While processors develop further, HwPCs will continue critical—not just for performance engineers, but for anyone wanting to learn about the intricate dance of transistors that powers modern computing.

Part I
Checking Hardware Compatibility

In this part, we'll describe how to read precise informations about our CPUs, and go beyond the `cpuid` linux command. We'll also cover how to find the events mesurable by the CPU.

3 Checking CPU capabilities : the `CPUID` instruction

`x86` architecture CPUs provide an instruction called `CPUID` that provides various informations on the CPU by reading special registers, for exemple, CPU name, memory address sizes, but also Hardware Performance Counters capacities.
The `CPUID` instruction is divided in `leafs` and `sub-leafs`, allowing to read multiple informations, stored statically in the CPU. These data are then stored in eax, ebx, ecx, and edx registers for the programmer to read.
For exemple, on intel, the leaf 0x80000008 stores in the eax register the physical address size and virtual address size of the CPU.
Intel and AMD manuals use a specific notation for this instruction which is the CPUID[LEAF].REG notation, used to facilitate the identification of the specific leaf and register from which a piece of information is retrieved. With the previous example, this notation would represent the value as CPUID[0x80000008].EAX.
A small tool was made for the project to read the CPUID instruction for intel and amd cpus, you can find it in the project's GitHub repository in the folder `tools/cpu_features_analysis`.

3.1 CPUID to find HwPCs informations on INTEL CPUs

`CPUID[0x0A].EAX` gives us these informations [Int, 2024a, Chapter 21.2]:

byte [7:0]: version number of the Architectural Performance Monitoring
byte [15:8]: number of general purpose Performance Monitoring Counter (PMC) per logical core
byte [23:16]: bit size of the PMC registers
byte [31:24]: number of architectural events
but, `CPUID[0x0A].EDX[7:0]` gives us also the number of fixed PMC per logical core, the fixed PMC, numbered from from 0 to n are tied to a specific event in the following table:

0	UnHalted Core Cycles	7	TopDown Slots
1	Instruction Retired	8	Topdown Backend Bound
2	UnHalted Reference Cycles	9	Topdown Bad Speculation
3	LLC Reference	10	Topdown Frontend Bound
4	LLC Misses	11	Topdown Retiring
5	Branch Instruction Retired	12	LBR Inserts
6	Branch Misses Retired		

Table 1: Fixed events

Here are the results on the Intel test machine :

```
Performance monitoring version                      5
Bit width of an IA32_PMCx MSR                       48
Number of general purpose PMC per logical core      8
Number of fixed PMC                                 4
Number of architectural events                      8

PREDEFINED ARCHITECTURAL EVENTS :
UnHalted Core Cycles (umask=00H,event=3CH)          supported
Instruction Retired (umask=00H,event=C0H)           supported
UnHalted Reference Cycles (umask=01H,event=3CH)     supported
LLC Reference (umask=4FH,event=2EH)                 supported
LLC Misses (umask=41H,event=2EH)                    supported
Branch Instruction Retired (umask=00H,event=C4H)    supported
Branch Misses Retired (umask=00H,event=C5H)         supported
Topdown Slots (umask=01H,event=A4H)                 supported
```

3.2 CPUID to find HwPCs informations on AMD CPUs

On AMD, it's `CPUID[0x80000001].ECX` that gives us informations about HwPCs [AMD, 2020a, Chapter 13.2.2] :
bit 10 : support of IBS (specific to AMD, explained in Section 7.2)
bit 23 : support of 6 Core Performance Counters.
bit 24 : support of 4 NorthBridge Performance Counters.
bit 25 : support of 4 L2 Cache Performance Counters.
The Core Performance Counters measures data about core elements of the cpu, like cycles, instructions etc...
NorthBridge in PC architecture was initally the part of the chipset that was responsible for high-speed communication between the cpu and components like the RAM, the PCI-express (for extensions like graphics cards), L3 cache, etc... This functionality was then moved into the CPU itself, but the naming convention stayed in performance monitoring, so NorthBridge counters measures data relative to transfer of data.
Here are the results on the AMD test machine :

```
6 Core Performance Counters
4 NorthBridge Performance Counters
No L2 Cache Performance Counter
IBS supported
```

4 List mesurable events

Each CPU architecture offers a specific set of events that its PMU can measure. To list the mesurable events on a specific CPU, we have several methods, and each method might not show all the events available.

Linux Sysfd Interface The directory `/sys/devices/cpu/events` shows a subset of events measurable. Each file in this directory corresponds to a measurable event, with its code and umask.

`perf list` **command** The command `sudo perf list` checks into the Linux perf subsystem to return a comprehensive list of hardware events supported by the kernel for the current CPU.

`Libpfm` **library** The `libpfm` library offers the most comprehensive and architecture-aware way to list measurable events. See more details for the `libpfm` library in Section 6.2. A program made with the library was made for the project and is present under the `tools/event_listing` directory.

Part II
HwPCs on x86 processors

In this part will be explained what compose Hardware Performance Counters (HwPCs) and how to use them. Explainations hold for `x86` processors, as it is the main focus of the project, these informations might not hold for other architectures such as ARM for exemple.

5 The hardware

In this section we'll explore the underlying hardware behind the Hardware Performance Counters, what compose what is called the `Performance Monitoring Unit` (PMU)

5.1 Store the HwPC data : Model-Specific Registers

Model-Specific Registers (MSRs) are special registers used for debugging, tracing, performance monitoring and toggling certain CPU features Alan Cruse [2006]. They started as simple test registers for the TLB on older Intel cpus, and they have their own instructions : `RDMSR` and `WRMSR`, which allows to read and write into these MSRs.
The MSRs used for HwPCs are `Event Select` and `Performance Monitoring Counter` (PMC) MSRs, they are both 64-bits wide on most recent CPUs. The Event Select MSRs contains all the necessary informations and parameters to select and start the measurements, and each Event Select MSR is linked to its own PMC MSR, if the event "x" is asked by Event Select MSR 1, the results of the said event measurements will be stored into PMC MSR 1.

5.1.1 Event Select MSRs composition : the exemple of Intel

We will now see how the Event Select MSR is composed through the exemple of the Event Select MSRs on Intel's CPUs, we'll not focus on AMD since it's nearly the same composition [Int, 2024a, Figure 21-6] [AMD, 2024, Chapter 2.1.14.3 `MSRC001_020[A,8,6,4,2,0]`].

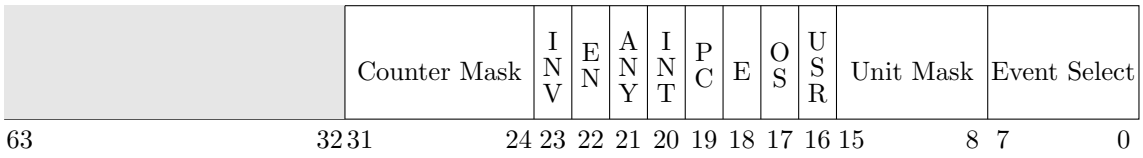

Figure 1: Bit Repartition of `IA32_PERFEVTSELx`

Event Select: It's the byte where we put the event's code.
UMASK: It's the byte where we put the event's umask.
USR: Controls whether we measure user events or not (corresponds to ring 3 level)
OS: Same as USR but for kernel and os-level instructions (corresponds to ring 1 level)
E: Edge Detect – Counts only when the event's signal transits from 0 to 1.
PC: Pin Control – Deprecated, used initially for Pentium and early Core era.
INT: Interupt – Interrupt the measure when the associated Performance Counter overflows.
ANY: Any Thread – Enable multithreading

EN: Enable – Enable/Disable associated Performance Counter.

INV: Invert – Inverts the Counter Mask.

CMASK: Allows to add some filters to the measurements.

Bits from 32 to 61 will be used later in the Architectural Performance Monitoring Version 6.

5.1.2 MSR addresses

To use a register, we need to know their address, for exemple, the WRMSR instruction takes the registers EAX, ECX and EDX, and writes EDX:EAX into the MSR ECX.
In Intel cpus [Int, 2024b, Table 2-2], `0x186` corresponds to the first Event Select MSRs, called `IA32_PERFEVTSEL0`, its corresponding Performance Counter MSR, called IA32_PMC0 holds the address `0xc1`. If we wanted to use another counter, we would use `0x187` and `0xc2`, increasing each time we want another counter.
For AMD cpus, there is a distinction between the types of events the MSRs can mesure, we will focus on the Core counters (for cycles, instructions etc.), which are named `MRC001_020[0..A]` for the Event Select MSRs and `MRC001_020[1..B]` for the Performance Counter MSRs, their name correspond to their address, so, if we want to use the first performance counter, we will use `MSRC001_0200` with `MSRC001_0201`, and their address are `C0010200` and `C0010201`.

5.2 Encode Events

We saw how we could store the measurement, and store the events' parameters, now, we need to know what can we put inside the fields "Event Select" and "UMASK".
Each event is associated to a `code` and a `umask`, which are hexadecimal values. The code defines a group of events, and the umask defines a subgroup of events, for exemple, on intel cpus, the group of event `"UnHalted ... Cycles"` is defined by the code `3C`, and the subgroups `"Core"` and `"Reference"` are defined respectively by the umasks `00` and `01`.
So to select the event `"UnHalted Core Cycles"`, you would have to enter the code `3C` with the umask `01`.

5.3 Start Mesuring

To start mesuring an event, the "low-level" method consists of writing the necessary informations (Event Code, UMASK, etc....) into the Event Select MSR to which event to measure and the measurements parameters, then, setting the Enable (EN) bit to 1.
As we saw earlier, to write into the ES MSR, we need to use the `WRMSR` instruction, and the measurement starts when the EN bit is set to 1, as soon as the `WRMSR` instruction finishes.
To stop, it is as simple as starting, set the EN bit to 0, you will then have the occasion to read the performance counter MSR with the instruction `RDMSR`
This method is relatively inconvenient, because it requires knowing many informations about our CPU architecture, the address of the Event Select MSR, the exact informations of the events, etc...
Check Appendix A and Appendix B to see how to use assembly to start counters.

5.4 How the CPU measures the events

Now that we know how the cpu encodes events and how to start mesuring, let's focus on how the PMU counts the events in the processor.
However, the exact internal implementation of hardware performance counters (HPCs) in modern

processors is not publicly documented in full detail. These implementations are considered proprietary and are likely protected as trade secrets. Informations below comes from hints from articles like Yasin, A. (2014). *A Top-Down Method for Performance Analysis and Counters Architecture.* ISPASS 2014, and are moslty theories.
When the cpu works, it sometimes encounter internal events (cache hit/miss, tlb hit/miss, branch taken/mispredicted, ...) which raises internal signals, that go through the `Central Event Bus`. With the Event Select MSRs, we defined the event that we wanted to measure, this data is given into a multiplexer, that gives the signal to our Performance Counter, which increments by 1 its counter.
But, sometimes, the counter might not be wide enough to contain the data, we call this an `overflow`, which can trigger the Performance Monitoring Interrupt (PMI), which enables cpu-special features like PEBS on intel.

6 The software

6.1 Linux Perf API

The Linux Perf API provides an abstraction of the "low-level method" provided in Section 5.3, it's what the `perf` command uses and most of the program uses to work with HwPCs, the library provides data structures and functions to measure CPUs events. The most notable function is the os call `perf_event_open()`, which takes several parameters such as the event configuration (code, umask etc...), the cpu to monitor, the process to measure, etc...
With the function, the kernel creates a `perf_event` object in memory, links this object to a PMU and then returns a file descriptor linked to the PMU. The file descriptor acts as a `userspace` which can interact with the kernel, we can then read the file descriptor to get the data after the measurements.
To enable or disable a counter, we can use the os call `ioctl`, which manipulate special file descriptors, we can with this function:

- Enable the counter : `ioctl(fd, PERF_EVENT_IOC_ENABLE, 0)`
- Disable the counter : `ioctl(fd, PERF_EVENT_IOC_DISABLE, 0)`
- Reset the counter : `ioctl(fd, PERF_EVENT_IOC_RESET, 0)`

Now the final problem to start mesuring the events is not putting the data and parameter, but to know what informations to put in, like the code, the umask,...

6.2 Libpfm

`libpfm` is a C-library that allows to measure events without knowing too much about performance counters, because the library has a translation system that converts an event name into a code and a umask.
It also replaces most of the Linux Perf API's functions and data structures to work more efficiently and simply, this library allows to sample a specific part of the code, like a loop or a function, rather than the whole program, like we have with the perf command.
The necessary components of the library are the following :

perf_event_attr Struct that stores the parameters of the event.

pfm_perf_encode_arg_t The struct that allows the final translation of the perf event name into a valid data, it contains a `perf_event_attr` variable.

pfm_initialize() This method is used to initialize the library, it's at this step that the library detects the underlying hardware and software configuration. Based on the informations, certain PMU supports are enabled.

pfm_get_os_event_encoding(event_name, PFM_PLM3, PFM_OS_PERF_EVENT, &arg) The method that takes the event name and put the necessary information into the `pfm_perf_encode_arg_t` variable `&arg` . The constants in the function call are :

PFM_PLM3 Measure at privilege level 3. This usually corresponds to user level. On `x86`, it corresponds to privilege rings.

PFM_OS_PERF_EVENT Tells the function to use the `linux perf api` to encode the event.

It even provides its own `perf_event_open()` function.

7 Intel and AMD CPUs specific features

7.1 PEBS (intel)

Earlier, we talked about PMI and the fact that it could enable special cpu features, PEBS is one of them [Int, 2024a, Chapter 21.6.2.4]. PEBS stands for Processor Event Based Sampling, and what it does is relatively simple, when a PMI is raised, the PEBS unit registers in the PEBS buffer a lot of informations about the CPU, like Instruction Pointer, general purpose registers values, memory adresses, ... The PEBS buffer is stored in a special place in the main memory (RAM).
This system is quite limited, as for the Architectural Performance Monitoring Version 5, only 4 `IA32_PMCx` (name for Performance Counter MSRs on intel) are compatible with this system, and not all event are compatible with it too.
To select which `IA32_PMCx` will measure with PEBS, we have to look into the `IA32_PEBS_ENABLE` MSR, and enable the bit corresponding to it. And, finally, if we enable PEBS for one MSR, in its corresponding IA32_PERFEVTSELx, we have to disable the bits AnyThread, Edge, Invert and CMASK.

7.2 IBS (AMD)

Instruction Based Sampling is a measuring method exclusive to AMD CPUs, it gathers specific metrics related to processor instruction fetch and instruction execution activity [AMD, 2020a, Chapter 13.3]. Instruction fetch sampling provides data about instruction address translation look-aside buffer (iTLB), and instruction cache (iCache) for a randomly selected fetch block[1] and is under the control of the `IBS Fetch Control Register`. Whereas instruction execution sampling provides information about instruction execution behavior by tracking the execution of a single micro-operation (op) that is randomly selected, and is under the control of the `IBS Execution Control Register`. The `IBS Fetch|Execution Control` registers are seperated from the other MSRs concerning HwPCs, and they contain roughly the same data to control the measurements, with `IbsRandEn`, which enable or disable the randomization of the sampling, `Ibs{Fetch|Op}Val`, which tells if the last measurement is valid or not, etc.
Once the mesurement for the current fetch or op is complete, the hardware signals an interrupt. Then, the interrupt handler reads the data captured, and can save it, to finally re-enable the hardware to take the next sample.

[1] A fetch block is the range of bytes the cpu fetches to get the next instruction.

Part III
Benchmarks

The tests were run on two machines, one with an Intel CPU, and the other with an AMD CPU, let's check their architecture:

```
11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40 GHz
8 logical cores, 4 physical cores
Fedora Linux 40 (Workstation Edition)
L1 Cache : 1 per physical core, 48kB
L2 Cache : 1 per physical core, 1MB
L3 Cache : 1 for all physical cores, 16MB
Performance Monitoring Version : 5
Bit width of a PMC register : 64
Number of general purpose PMC per logical core : 8
Number of fixed PMC per logical core : 4
Number of architectural events : 8
```

```
AMD Ryzen 5 3600X 6-Core Processor
12 logical cores, 6 physical cores
Fedora Linux 40 (Workstation Edition)
L1 iCache : 1 per physical core, 32kB
L1 dCache : 1 per physical core, 32kB
L2 Cache : 1 per physical core, 512kB
L3 Cache : 1 for 3 physical cores, 16MB
6 Core Performance Counters
4 NorthBridge Performance Counters
No L2 Cache Performance Counter
IBS supported
```

8 Method

The benchmarks were done on two seperate files, first a CPU-bound benchmark containing only floating point operations and the second is a memory-bound benchmark, a matrix multiply program were the matrices' size goes over the size of the last cache so the data goes into main memory.
With these programs, we'll first do the benchmarks with the `perf` command, to do so, we simply execute the line `perf stat -e <list_of_events> ./<file_name>` and output the result into a .txt file. Then we extract the data with a simple python script, put the data into a .csv file to later plot it with R. Then, with libpfm the method isn't really different, we simply output the result into a csv, as we control how the data is outputed by the program.
You can check the benchmarks' codes in the Appendix C and Appendix D

9 Results

In these benchmarks, we can see the difference between the benchmarks done with the perf command line tool and the benchmarks done with the libpfm c-library. The key difference here is that the benchmarks done with libpfm don't take into account the initialisation of the program, which is really huge in the memory-bound benchmarks, as the initialisation requires to iterate over the two initial matrices.

Moreover, we can also see slight differences between Intel and AMD performances, which might come from differences in the test's environement, different ressource-management by the CPUs or even differences in the compiler, as `gcc` might optimize better for Intel cpus than AMD ones.

Also, we can notice that in Figure 3 AMD benchmarks didn't include L3 Cache measurements, as less recent AMD CPUs did not incorporate L3-Cache measurement (the output on perf command showed `not supported`).
We also saw that the AMD CPU should not have measured L2 Cache events, since there isn't any L2 Cache performance counters, but L2 Cache performances counters offers seperated measurement that are often more precise.

Now with these results we can know where we can reduce bottlenecks, on the memory-bound benchmarks [Figure 3], we can see that there is a huge amount of L1-cache hits AMD-side, so optimizing the code to reduce L1-cache usage seems a reasonable direction. CPU-bound benchmarks, however, there are just float computations, so reducing the number of instructions might be not so possible [Figure 2].

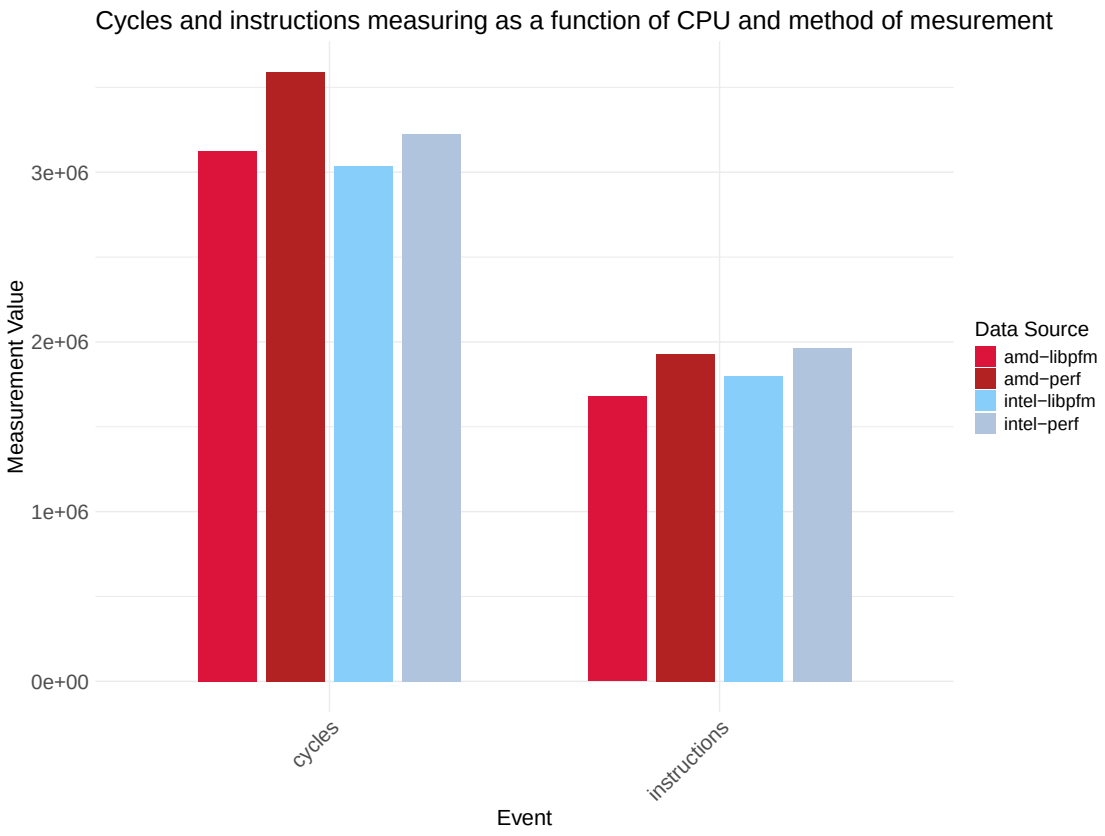

Figure 2: CPU-bound benchmarks

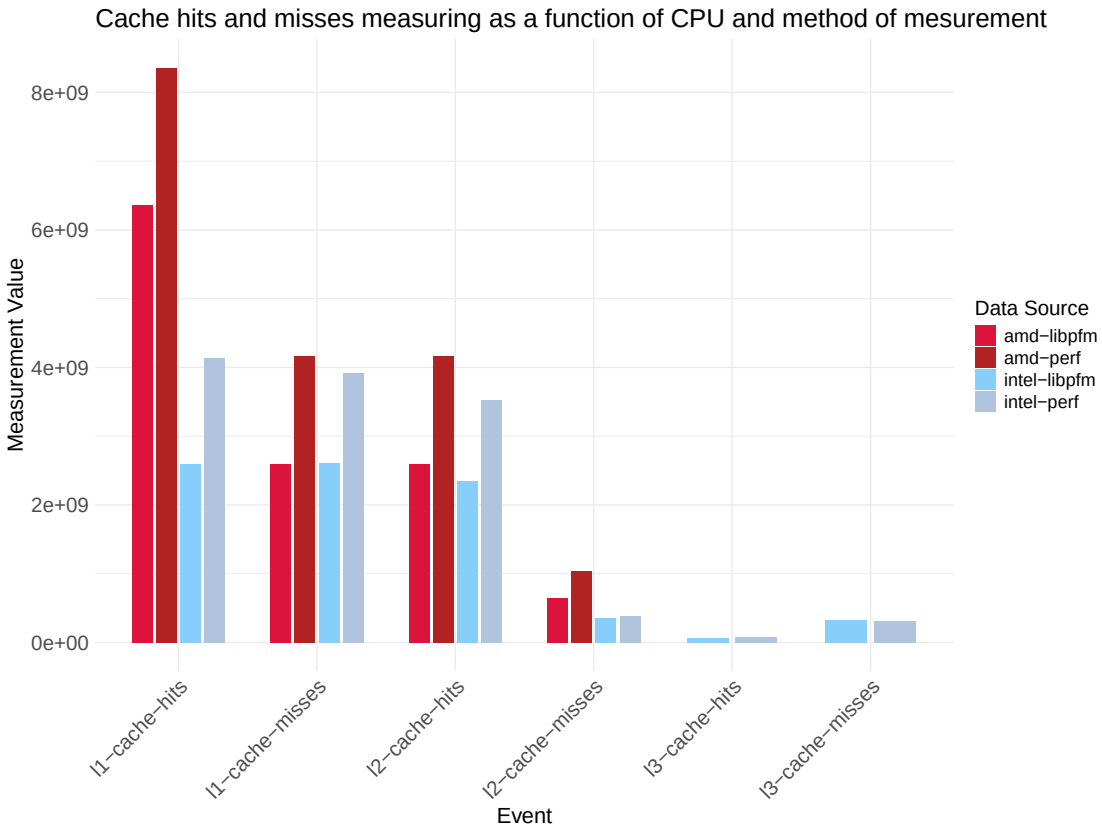

Figure 3: Memory-bound benchmarks

Part IV
Conclusion

10 Discussion

Hardware Performance Counters (HwPCs) are a low-overhead, powerful interface to observe and analyze the internal operations of modern CPUs. Throughout this project, we learned about both the theoretical foundations and real implementations of HwPCs on `x86` architectures, with a focus on Intel and AMD processors.
We began by identifying CPU features using the CPUID instruction and investigating MSR layout and configuration. This gave us an understanding of how events are encoded and measurements are initiated at the hardware level. We then proceeded to software interfaces such as the Linux perf API and the libpfm library, which simplify interaction with performance counters and provide high-level access to a large number of measurable events.
To validate our analysis and toolchain, we developed CPU-bound and memory-bound benchmarks. Benchmarks confirmed HwPCs' ability to distinguish between different performance bottlenecks and provided us with an insight into actual program behavior.
This project illustrates the powerful role HwPCs can play in performance debugging, compiler analysis, and low-level software optimization. In an age where performance tuning is more important than ever, HwPCs remain one of the most precise and underutilized tools in a developer's kit.

11 Future Work

Several directions can be explored to expand upon this work. First, incorporating energy-related performance counters, as they would underline the compromises between performance and energy consumption. This is particularly relevant for embedded systems and mobile devices, where energy efficiency is often as critical as raw performance. Exploring how optimization levels or hardware features impact energy usage, in combination with execution time, could lead to more sustainable computing practices.
Additionally, integrating security-related considerations into performance counter analysis presents another interesting direction. Certain classes of side-channel attacks exploit patterns in branch prediction, cache usage, or execution time—often detectable through hardware counters. By profiling and analyzing such patterns, we could potentially identify anomalous behaviors or assess the susceptibility of various programs or systems to specific attack vectors. This would be especially relevant in contexts like cloud computing or multi-tenant environments where security is crucial.
Finally, automating the counter selection and analysis pipeline could further improve the accessibility and reproducibility of experiments. Machine learning approaches might also be employed to correlate counter trends with broader software behaviors or system states, enabling smarter compiler heuristics or runtime decisions.

Appendices

Assembly code to use HwPCs

```
; values selected for an Intel CPU
mov ecx, 0x186          ; select IA32_PERFEVTSEL0
mov eax, 0x004300c0     ; low 32 bits of value (event + config)
mov edx, 0x00000000     ; high 32 bits of value (here: 0)
wrmsr                   ; write to MSR[ECX] = EDX:EAX

mov ecx, 0xc1   ; Move the MSR address into ECX
rdmsr                   ; Read MSR at ECX into EDX:EAX
```

Assembly code to use HwPCs in C

```
intel:
    __asm__ __volatile__("wrmsr" : : "c" (0x186), "a" (0x004300c0));
        // if needed, you can also set the edx register to set the 32 highest bits
    __asm__ __volatile__("rdmsr" : "=a" (a), "=d" (d) : "c" (0xc1));
        // d variable holds the 32 highest bits, and a the 32 lowest.

amd:
    __asm__ __volatile__("wrmsr" : : "c" (0xc0010200), "a" (0x004300c0));
        // if needed, you can also set the edx register to set the 32 highest bits
    __asm__ __volatile__("rdmsr" : "=a" (a), "=d" (d) : "c" (0xc0010201));
        // d variable holds the 32 highest bits, and a the 32 lowest.
```

CPU-bound benchmark's code

```
for (i=0; i< N; i++){
    x=x+y;
    y=x+t;
    z=x+z;
    x=t+y;
    t=y+x;
}
```

Memory-bound benchmark's code

```
for (int i=0; i< N; i++){
    for (int j=0; j< M; j++){
        for (int k=0; k< P; k++){
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}
```

References

libpfm(3) — linux manual page, 2010. URL https://www.man7.org/linux/man-pages/man3/libpfm.3.html.

perf_event_open(2) — linux manual page, 2024. URL https://www.man7.org/linux/man-pages/man2/perf_event_open.2.html.

Alan Cruse. Lecture notes, advanced microcomputer programming class, university of san francisco, 2006. URL https://www.cs.usfca.edu/~cruse/cs630f06/lesson27.ppt.

AMD64 Architecture Programmer's Manual, Volume 2: System Programming. AMD64 Technology, 2020a.

AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions. AMD64 Technology, 2020b.

Processor Programming Reference (PPR) for AMD Family 17h Model 71h, Revision B0 Processors. AMD64 Technology, 2024.

Intel® 64 and IA-32 Architectures Software Developers Manual, Volume 3 (3A, 3B, 3C & 3D): System Programming. Intel Corporation, 2024a.

Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 4: Model-specific Registers. Intel Corporation, 2024b.