Karlsruhe Institute of Technology

Lightweight Virtual Machine Introspection for Adaptive Energy Estimation on Multicore x86 Systems

Bachelor's Thesis
submitted by

Tobias Merkel

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Dipl.-Inform. Thorsten Gröninger

24. June 2025 – 24. October 2025

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, October 24, 2025

Abstract

Cloud providers restrict access to fine-grained energy metrics in multi-tenant virtualized environments, as they can be exploited for side-channel attacks. Meaningful interpretation requires system-wide oversight, which is prohibited for guest VMs. Developers lack the tools to debug and optimize applications for energy consumption under these restrictions. As a result, energy statistics should be provided by the host system. This poses a semantic gap, as the host needs to introspect the guest to calculate meaningful energy statistics. Existing approaches are either invasive and require VM modifications, or attribute energy only to the entire VM.

In this thesis, we address this problem by presenting two introspection approaches that provide metrics required for energy attribution. Both techniques leverage hardware features, namely *MOV to CR3 VM exiting* and *Intel Processor Trace (Intel-PT)*. The *MOV to CR3 VM exiting* approach allows us to trace an application start and execution time, happening in the same address space, on modern AMD and Intel processors. *Intel-PT* enables high-resolution process switch detection and precise cycle counting through the use of trace packets and timestamp counter estimation. We evaluate Intel-PT for its impact on performance and power consumption, identifying optimal configuration parameters. Our experiments indicate an increase in execution time within the range 0 - 6 % and a minimal power consumption overhead of 2 to 3 % in the worst case.

This could in the future allow for better energy attribution in data centers, without the need to modify guest operating systems, yet identify energy-wasting processes.

Contents

Chapter 1

Introduction

Cloud providers export estimated greenhouse gas emissions on a monthly basis to their customers [1–4]. If consumers want to reduce greenhouse gas emissions at the software level and improve the energy efficiency of their programs, these providers do not have the proper tools to report energy consumption on a fine-grained level.

Power-metric interfaces such as RAPL enable fine-grained measurements at the system or package level, but lack support for individual processes, VMs, or containers. As there are multiple tenants on such a system, cloud providers disable such reporting due to the possibility of side-channel attacks on other tenants [5, 6].

To enable energy reporting, there are some approaches shown, e.g., by EnergAt [7], VMware [8], and Kepler [1, 9]. However, their work either reports estimated energy consumption for a whole VM without process attribution or violate transparency. For instance, Kepler relies on eBPF programs, which need to be injected into the VM.

To overcome these issues, we envision a debugging framework that inspects a virtual machine including running applications in the background with minimal overhead and only employs more aggressive logging and inspection on user interaction or predefined events.

We have built a prototype which allows estimated energy consumption / power reporting at the process level while bridging the semantic gap introduced by hardware-accelerated virtual machine boundaries, though certain features are not yet fully operational. We envision a system that can also be used to gather valuable data for other purposes, e.g., memory tracing or debugging, keeping the impact minimal for running VMs.

Our work leverages Intel Processor Trace (Intel-PT), a hardware feature to trace the execution context and performance of a process or the whole system with minimal overhead. Intel-PT allows keeping track of the currently running

process by tracing modifications to the CR3 register, which holds a reference to the page table in use, on x86. Furthermore, timings, passed clock cycles, and, depending on the microarchitecture in use, information about power states are also included in the trace.

By utilizing this tracing mechanism, we are able to generate a timed trace of running processes. Combined with data from the (virtualized) Performance Monitoring Unit (PMU) and by utilizing Kepler's energy attribution framework, we ideally are able to generate a real-time process energy consumption estimation, even for processes running inside VMs. This data can, in a next step, be mapped to human-readable process information, i.e. PIDs, using noninvasive or minimally invasive virtual machine introspection. A possible speed-up would be a shadow VM / process to gather information from within the VM, without the need for interruption or code insertion.

We might also export these metrics using VMSH [10] to the guest, if it opts in, to receive such information in the future. Unknown human-readable processes are exported by the translated CR3 address. This occurs while using nested virtualization or stale mappings of page table addresses to process identifiers. The whole accounting process is OS-agnostic and only relies on information gathered with minimal—additional—interruption of a running VM. Introspection capabilities are out-of-focus for our prototype, but may be limited to common cloud operating systems, like Linux.

Furthermore, we compare the performance implications with a different approach, which uses MSR and CR3 register write interception. This technique enables the use of AMD hardware that does not have an Intel-PT counterpart.

We give an introduction of the used technologies in Chapter 2, a brief overview of related work in Chapter 3. Our design can be found in Chapter 4, followed by the description of our research prototype in Chapter 5 and its evaluation in Chapter 6.

Chapter 2

Background

The main technology used for our introspection and energy reporting approach are tracing and performance monitoring capabilities of modern Intel x86 processors.

2.1 Virtual Machines

One of the main goals of operating systems is to provide each process with the illusion that it is the only process that runs on hardware. Thus, it has to virtualize hardware in one way or another.

Virtual machines extend this concept to whole operating systems; extending hardware virtualization to a state where operating systems (guests) run simultaneously on a single machine. Doing so provides multiple advantages, e.g., isolation, or ease of debugging / testing of applications on multiple emulated architectures and operating systems.

Virtual Machine Monitors (VMMs), also called hypervisors, are responsible for managing virtual machines. They either run directly on the hardware (type 1) or on top of an operating system (type 2) [11]. Major cloud providers [12] resort to type 2 hypervisors, backed by KVM.

Akin to context switching, CPU manufacturers provide mechanisms to enter (VM entry) and exit (VM exit) virtual machines [13, 14]. On exit, the processor saves and restores the processor state and execution context, e.g., all general-purpose registers.

2.1.1 Virtual Machine Control Structure

Prior to entering a virtual machine, the CPU and ultimately the VMM has to set up structures to organize VMs. Moreover, a dedicated space for the information saved on VM exits is required.

Intel implements this structure with the Virtual Machine Control Structure (VMCS) [15]. AMD provides the Virtual Machine Control Block (VMCB) [14], both serve the same purpose.

Notably, VMCS provides VM entry and exit control fields which can be used to back up and restore MSRs. It is possible to utilize this feature to restrict Intel-PT trace generation to the guest.

Furthermore, it specifies event interception, e.g., interrupts, exceptions, and instructions.

2.1.2 Introspection

Virtual Machine Introspection (VMI) refers to the process of extracting information of VMs from the outside. It is mainly used by security researchers who want to investigate how malware behaves while keeping an additional layer of security. VMI can be separated into active and passive introspection [11]. Passive introspection refers to unsynchronized observation without interfering with the VM, whereas active introspection intercepts events. Such events range from accessing registers to accessing memory or placing breakpoints.

Other applications of VMI are resource-aware scheduling [16] and VM intrusion detection and prevention [17].

These techniques can be used to access additional data to bridge the semantic gap between the host and the guest OSs.

2.2 Running Average Power Limit

Both Intel and AMD report energy consumption for the package domain through the Running Average Power Limit (RAPL) MSR. Values are typically updated in a regular time interval, i.e., every $1\,\mathrm{ms}$ on Intel CPUs. AMD also exports power consumption for each core [18], whereas Intel provides information for DRAM energy consumption and uncore components [13], i.e., the graphics unit depending on the model.

RAPL is a widely utilized mechanism in the field of energy estimation and accounting (Section 3.2).

2.3 Intel Processor Trace

Intel Processor Trace (Intel-PT) [15] allows tracing of programs and the whole system with minimal overhead. It follows the goal to trace only the bare minimum required to reconstruct the execution in combination with the executed object (process, container, VM).

Intel-PT is an extension for the Intel x86 architecture and has been introduced with the Broadwell microarchitecture with the goal of atomically writing identifying information about the execution to main memory or a platform-specific tracing subsystem. The captured data provides, depending on the architecture, insight for:

- The currently running application
- The control flow
- Timing
- Power state changes
- Interrupts and exceptions
- VM entries and exits

Furthermore, user defined information can also be added to the trace (using a new instruction `PTWRITE`). It is to be noted that the supported features vary depending on the deployed microarchitecture, e.g., power state changes are currently only available for low-power architectures used for E-cores.

Intel provides a reference implementation for decoding traces [19]. There also exist several tools [20, 21] for capturing as well as decoding traces in Linux user space.

2.3.1 Data Output

Intel-PT generates packets for each event, e.g., writes to the CR3 register, which points to the page table of a process, or periodically, e.g., for system time information. These packets are either written to the main memory while bypassing caches or forwarded to a tracing device. Intel-PT supports three ways to collect and make the data available to the user:

Firstly, a single physically contiguous memory range that acts as a circular buffer.

Second, multiple physically contiguous memory ranges which are treated as one big memory range. It is described in detail in Section 2.3.1(Table of Physical Addresses).

Finally, a platform-specific tracing subsystem.

Once a memory range is filled, Intel-PT can trigger a Performance Monitoring Interrupt (PMI) if requested.

For each core / hyperthread with tracing enabled, a separate output region must be specified.

Table of Physical Addresses

Multiple ranges of physically contiguous memory can be linked together to form a single large continuous trace buffer range from *Intel-PT*'s point of view. To do that, Intel provides a table structure. Each entry within the table refers to one physically continuous range. An entry consists of the physical address of the range, the size encoding of the range, a stop bit, an interrupt bit and an end bit.

If the *end* bit is set, it indicates that the address points to the next table base. The table may have its own address written in it, creating a circular trace buffer. The *stop* as well as the *interrupt* bit must be set to 0 in case the *end* bit is set. Otherwise, a ToPA error occurs.

If the *stop* bit is set, the hardware disables trace generation and sets a *stopped* bit within the `IA32_RTIT_STATUS` MSR once the corresponding range is filled. This mechanism is used to avoid accidental overwriting of unconsumed data. Ranges marked with the *interrupt* bit cause a non-maskable interrupt to be issued after the range has been filled. Once an interrupt is issued, the hardware sets the `IA32_PERF_GLOBAL_STATUS.Trace_ToPA_PMI` bit. In contrast to ranges marked with the *stop* bit, tracing continues even while handling the interrupt unless tracing is manually disabled. It is legal for the *stop* and *int* bits to be set at the same time.

Prior to enabling tracing with ToPA, the `IA32_RTIT_OUTPUT_BASE` MSR, which holds a physical address to the currently active ToPA, must contain a proper address. `IA32_RTIT_OUTPUT_MASK_PTRS` stores an entry index of the ToPA and an offset within the entry to which the trace hardware begins to write packets. Once tracing has stopped, either by hitting a *stop* bit within the ToPA or by manually clearing `TraceEn`, `IA32_RTIT_OUTPUT_BASE` and `IA32_RTIT_OUTPUT_MASK_PTRS` are updated by the hardware to reflect the current position within the trace buffer. Unfortunately, both MSRs may not reflect the current write position during tracing.

2.3.2 Packets

Packets consist of a header for packet identification and, optionally, a payload.

In Table 2.1 we introduce some of Intel-PTs packets. Packets TNT to PSB are available on recent processors and provide insight into the basic execution. The `PTWRITE` instruction that yields a PTWRITE packet allows adding user defined

data to the trace as well as power packets (EXSTOP-PWRX) that offer details on sleep and power transitions, have been introduced with the Goldmont Plus microarchitecture [15]. Processor Event Based Sampling (PEBS, documented in the Intel Architectures Software Developer's Manual (SDM) [13]) packets (BBP-BEP) are available starting with the Tremont microarchitecture. Event packets (CFE-EVD) that expose information about asynchronous events has been introduced with the Gracemont microarchitecture. However, these features might not be available even on cores with the required microarchitecture. Support must be validated using CPUID [15].

We use the blue highlighted packets to identify when and for how long an application runs. This information allows us to correlate the trace with measured energy consumption, e.g., RAPL. Green packets aid in energy estimation by offering insights into the processor's state and performance. Finally, red packets provide detailed information about the executed process state and are therefore used for advanced debug reporting, or could be used for fine-grain thread analysis.

Packet	Description	Limitiations
Taken/Not-taken (TNT)	Contains whether the last N direct branches have been taken.	IP must be resolved with other packets and the executed code.
Target IP (TIP)	Encodes the jump destination, including VM entry / exit and interrupts.	Is not generated for direct branches.
Packet Generation Enable (TIP.PGE)	Contains the IP address upon enabling tracing.	
Packet Generation Disable (TIP.PGD)	Contains the IP address upon disabling tracing.	
Flow Update (FUP)	Provides the source IP of jumps caused by asynchronous events, e.g. interrupts or VM exits.	

Paging Information (PIP)	Generated on CR3 changes and contains the updated page table address. Used to keep track of the currently running application. Contains a non-root bit. It is set if the guest is executing.	Cant be associated with VM without further tracking.
TraceStop	Generated when IP is out of scope and Intel-PT should stop tracing.	
Core:Bus Ratio (CBR)	Reports the current core to bus frequency ratio. Can be used to correlate cycles, frequency and time.	
Timestamp Counter (TSC)	Contains a copy of the timestamp counter at the point of packet generation. This includes the software offset, which may differ in VMs.	Is not generated often compared to other packets.
Mini Time Counter (MTC)	Periodically reports a common timestamp copy, i.e., the core's crystal clock tick count. This is used to correlate real time with generated packets.	Has a lower resolution than the TSC payload.
TSC/MTC Alignment (TMA)	Generated in combination with a TSC packet and contains the MTC payload & normalized number of processor cycles since the last crystal clock tick. Used to correlate TSC, MTC and processor cycles.	
Cycle Count (CYC)	Encodes the passed clock cycles since the last packet of its kind. It is used to count passed clock cycles.	
VMCS	Encodes VMCS address upon a `VMPTRLD` instruction. This must happen before a vCPU can be scheduled. Can be used for keeping track of the currently running VM.	It does not track VM entries.
Overflow (OVF)	Indicates that a buffer overflow occurred. Holds the IP at which tracing has been resumed.	

Packet Stream Boundary (PSB)	Is a feedback indicator after a fixed output of bytes written to the trace buffer.	
Block Begin (BBP)	Indicates the begin of PEBS data.	Might be unavailable on some processors / cores.
Block Item (BIP)	Data of PEBS event.	Might be unavailable on some processors / cores.
Block End (BEP)	End of PEBS event data.	Might be unavailable on some processors / cores.
PTWRITE (PTW)	4 or 8 bytes of data added to the buffer caused by an executed `PTWRITE` instruction. It is used to manually add user defined data to the trace.	Might be unavailable on some processors / cores.
Execution Stop (EXSTOP)	Indicates that the execution has stopped.	Does not specify the exact event but indicates a change in frequency. Might be unavailable on some processors.
MWAIT	Encodes the CPU power state (C-State) changes caused by `MWAIT` alike instructions.	Might be unavailable on some processors / cores.
Power Entry (PWRE)	Is sent on C-State entries.	Might be unavailable on some processors / cores.

Power Exit (PWRX)	Generated on C-State exits.	Might be unavailable on some processors / cores.
Control Flow Event (CFE)	Indicates the occurrence of asynchronous events, i.e., interrupts or exceptions. Can be used to detect VM entries, exits and the interrupt type if it caused an exit.	Might be unavailable on some processors / cores.
Event Data (EVD)	Includes information about CFE packets which were generated due to a page fault. Is used to get information about the faulted address, VM exit qualification and reason.	Might be unavailable on some processors / cores.

Table 2.1: Intel-PT packet definitions

2.3.3 Configuration

The `IA32_RTIT_CTL` Model Specific Register (MSR) is available per core or thread if the microarchitecture is capable of hyperthreading. It is used to enable tracing and control trace packet generation.

In the following, we highlight important configuration options available in the MSR. Each entry is annotated with the bit length of the field. Unless stated otherwise, a value of 1 indicates that the feature is enabled.

TraceEn:1 controls the trace generation.

OS:1 allows packets to be generated if the *Current Privilege Level (CPL)* equals zero. So all kernel activity can be traced. The CPL is used for access control to protected instructions. A CPL of 0 encodes kernel privileges.

User:1 allows packets to be generated if the CPL is greater than zero. As the name suggests, all user level packets are recorded.

CYCEn:1 enables the generation of cycle packets.

CycThresh:4 controls how many cycles have to pass until the next CYC packet can be generated. It is determined by the following equation [21]:

$$min\_cycles = 2^{CycThresh-1}$$

Not all values are supported on all platforms. Intel provides a bitmap of valid `CycThresh` values, which can be queried by the `CPUID` instruction. Each set bit positions indicate a valid input. For example, `0b1010` indicate that 1 and 3 are valid inputs, but 0 and 2 are not.

PwrEvtEn:1 controls power event packets (EXSTOP-PWRX in Table 2.1).

ToPA:1 indicates whether the ToPA mechanism described in Section 2.3.1 or a single output range is used.

MTCEn:1 enables Mini Time Counter packets.

MTCFreq:4 controls the frequency of how often the MTC packets are generated in respect of the Crystal Time Clock (CTC). The CTC ticks which need to pass until the next MTC packet is generated can be expressed with the following equation [21]:

$$mtc\_frequency = \frac{CTC\_freqency}{2^{MTCFreq}}$$

where the $CTC\_freqency$ is the frequency of the Crystal Time Clock. Akin to the `CycThresh` value, the `MTCFreq` has to match a bitmap provided by `CPUID`.

TSCEn:1 controls Time Stamp Counter packets.

BranchEn:1 enables the generation of packets which indicate the position within the executed program. This includes, but is not limited to TNT, TIP and FUP packets. It does not affect packets required to determine which process is currently running.

PSBFreq:4 describes after how many bytes written to the trace buffer a PSB packet should be generated. The number of bytes needed is seen in the next equation [21]:

$$psb\_period = 2^{PSBFreq+11}\ bytes$$

Again, the value must satisfy the requirements provided by `CPUID`.

EventEn:1 enables the generation of event packets (CFE-EVD in Table 2.1).

DisTNT:1 suppresses TNT packets even if `BranchEn` is set.

2.3.4 Performance Counter

In order to analyze execution performance, insight is required. While this can be partially archived with software, CPU vendors ship a *Performance Monitoring Unit* providing counter MSRs for a variety of events. Two types of performance counters exist on modern hardware, fixed-function and general purpose counters. Fixed-function counters are hardwired to a specific event such as retired instruction count or core cycles. In contrast, general purpose counters can be configured by the user. For instance, it can provide information about cache misses or branch instructions where no fixed-function counter exists [13]. On AMD platforms, only configurable counters are available whereas Intel platforms provide both [22].

Chapter 3

Related Work

In this Section, we summarize and discuss previous work related to Virtual Machine Introspection (VMI) and energy attribution.

3.1 Virtual Machine Introspection

As previously mentioned in Section 2.1.2, VMI is the process of extracting information from VMs. We present a range of introspection approaches that utilize agent-based techniques as well as external observation and communication across VM boundaries.

VMSH is a hypervisor-agnostic abstraction for attaching services to a VM on demand. Such services range from adding in-VM functionality to uni- and bidirectional communication between the VM and the host.

Thalheim et al. achieve this by implementing VirtIO character and block devices using rust-vmm [23], a common set of user space libraries for building Virtual Machine Monitors (VMMs) developed and relied upon by major cloud providers [24].

A minimal eBPF program reveals the location of the guest VM's memory. Using this information, a library is uploaded to the guest to make the services available. Prior to reentering the guest, the Instruction Pointer (IP) is updated to the sideloaded library's entry point using system call injection. VMSH uses ptrace to halt and hijack the VMM process. This is required as only the VMM process is allowed to perform modifications to the guest, which includes register altering. It continues by determining the Kernel Address Space Layout Randomization (KASLR) offset and resolving a minimal set of kernel symbols, which are used to start the drivers and make the VirtIO devices visible to the guest. For block devices, OverlayFS prevents

conflicts between the guest- and service-provided files. Inside the guest, access to the Memory Mapped IO (MMIO) of the VirtIO devices triggers a VM exit. These exits are intercepted with ptrace and forwarded to the VMSH device, which can proceed processing the exchanged data.

Their prototype implementation works with Linux v4.4 up to v5.10. However, porting it to a newer guest OS is expected to be of low effort [10].

LibVMI is an open-source library with the purpose of introspecting a VM's state from outside. LibVMI is designed to work with running VMs as well as memory snapshots. Furthermore, passive as well as active introspection in real-time is possible with a range of hypervisors, such as Xen, Bareflank, and QEMU-KVM. For QEMU-KVM, there are multiple choices: Using a patched QEMU-KVM system or KVM's debugging interface [16, 25, 26].

Libkvmi is an introspection library for KVM and QEMU akin to LibVMI, introduced in 2020 by Bitdefender.

The introspection subsystem for KVM (KVMi) exposes an introspection interface for each VM with a Unix domain socket. The user-space library builds an API on top of the socket [11].

Furthermore, Libkvmi has been integrated into LibVMI. Currently, only Linux v5.4 is supported and porting it to newer Linux versions is not feasible in a reasonable amount of time [27, 28].

VMIFresh introduces freshness guarantees for the memory, process, and address translation cache to LibVMI. It works by generating exceptions for write access on pages of interest. If an exception occurs, VMIFresh can update the cache entries. In comparison to the unmodified implementation of LibVMI, which simply flushes the caches every second or not at all [11], the cache contains no stale data entries. For instance, the process cache, which holds a mapping of process identifiers to the addresses of the page table, listens for memory access of pages containing `task_struct::mm`.

Furthermore, they propose an integration into Volatility 3 [29], which is a framework for advanced memory analysis with a focus on static memory dumps [30].

LibVMI for AMD & ARM summarizes the state-of-the-art for VMI systems for Intel, AMD & ARM architectures. Dangl et al. extend the VMI functionality available to Intel CPUs to AMD and ARM processors. Their approach leverages architectural structures, such as the Interrupt Descriptor Table (IDT) on x86 and the Vector Base Address (VBA) on ARM, to efficiently determine the KASLR offset. As these structures are defined within the

KASLR region, they are relocated. Furthermore, their relocated position is visible through privileged registers. Calculating the difference yields the offset in constant time.

Linux stores a list of processes in a linked list of `task_structs`. The first entry in this list is the `init_task`, which is also rebased with KASLR. In contrast, the previous algorithm performs a linear search of the address space for the `init_task` struct using an inverted page table . The found position is compared to the address without KASLR, thus exposes the KASLR offset.

The constant time algorithm poses a significant improvement over the previous approach, especially on ARM systems with a significantly larger search space [31].

VMware Tools provide a set of guest-installed services for Linux, macOS, and Windows which can be queried by the hypervisor to gain knowledge about the running VM. For instance, it is capable of acquiring a list of running processes inside the VM and report it to the hypervisor.

3.2 Energy Accounting and Estimation

In this Section, we examine the field of energy accounting.

Reliable Basic Block Energy Accounting studies fine-grained energy attribution for basic blocks, which are sequences of instructions without conditional branches or jumps. Before each basic block, code to trace RAPL (Section 2.2) via the Linux driver, is injected using compiler modifications. The same is done before external function calls and after returns. The consumed energy is retrieved via the Linux RAPL driver. With Intel Processor Trace (Intel-PT), the basic blocks are identified within external functions. Finally, the RAPL delta energy is attributed to the basic blocks by calculating the ratio of the time spent in the block and the RAPL update interval [32].

Kepler is a framework for energy attribution of processes in containers. It collects system-wide energy consumption data reported by RAPL, ACPI, and NVML. It measures idle, activation, and dynamic power consumption, and attributes energy usage to individual processes by correlating power data with process-level metrics. These metrics include CPU time, interrupts, and performance counters that are obtained with an eBPF module. Kepler trains power models based on the collected data to estimate per-process

energy usage. In their work, Amaral et al. evaluated the accuracy of models with different combinations of process metrics. [9, 33]

EnergAT provides fine-grained, thread-level energy accounting that respects multiple NUMA nodes and DRAM usage patterns across heterogeneous systems. The system performs accounting based on CPU time and memory usage specific to individual NUMA nodes [7].

Chapter 4

Design

In this chapter, we present the design of how existing hardware features, namely *Intel Processor Trace (Intel-PT)* and *CR3 write exiting* can be leveraged to provide introspection for process energy attribution. We present two approaches, each utilizing one of these mechanisms.

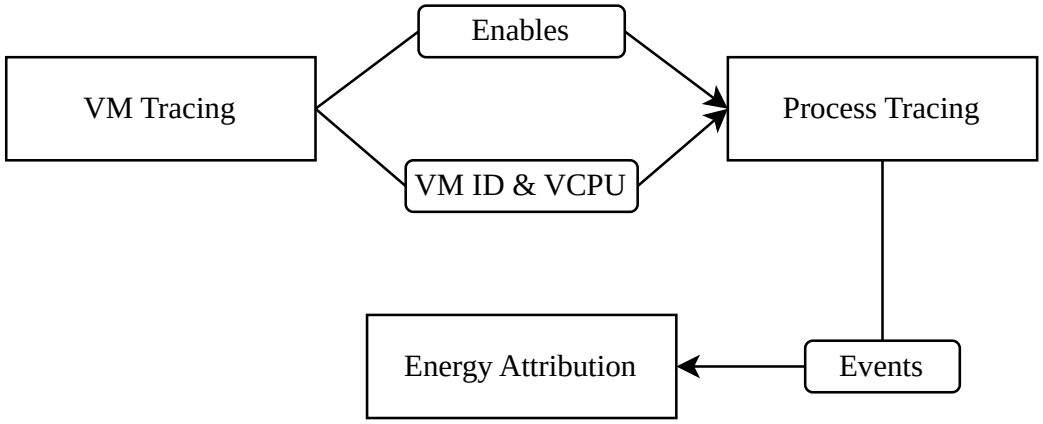

Figure 4.1: Overview of the main design components

The whole process comes down to three main components visualized in Figure 4.1:

VM Tracking tracks which virtual machines exist, assigning unique IDs to them. With the Intel-PT method, it tracks further information about vCPUs in order to enable identifying processes uniquely across host-guest boundaries. Upon a user-initiated enable request, the system is responsible for starting the *Process Tracing* component.

Process Tracing tracks when a process is running and collects information within the hypervisor, which can be used to draw conclusions about the

energy consumed. Furthermore, it gathers uniquely identifying process information. In our design, this is the page table address, which resides in the CR3 register if the process is currently running. We pass this information as a stream of events to the *Energy Attribution* component.

Energy Attribution measures overall packet energy consumption and data required to split up the tracked energy consumption and attribute it to processes.

4.1 VM Tracking

The *VM Tracking* component tracks created Virtual Machines (VM) since the boot of the host. A unique identifier is assigned to each VM on creation. Once the user requests enabling of energy estimation for a VM, it starts the *Process Tracing*.

Depending on the *Process Tracing* approach picked, further data is required to uniquely identify processes. The main problem is that VMs use their own virtual address space. Multiple guests are allowed to place the page table at the same address within their address space. This leads to duplicate CR3 content, thus purely relying on the page table address for process identification is infeasible. Such addresses are not unique and might further change over time as processes are created and destroyed.

On Intel platforms with Virtual Machine Extensions, each vCPU is paired with a control structure (VMCS) (Section 2.1.1). Prior to executing work on a vCPU, the VMCS has to be loaded into the physical CPU using the `VMPTRLD` instruction. One Process Tracing approach we present in our design takes the VMCS address as a hint for VM process identification. Thus, the *VM Tracking* component has to maintain a lookup data structure of the VMCS address and the corresponding vCPU and ultimately the VM.

Figure 4.2 shows how the lookup structure is maintained. Once the user creates a new VM, the hypervisor assigns a unique ID to it. After the identifier is established, the user may add one or more vCPUs to the VM. The hypervisor proceeds with creating the vCPUs, ultimately allocating memory for the VMCS. We take the physical address of each VMCS and pass it along with the VM ID and vCPU it to the *VM Tracker*.

If the user requests the hypervisor to destroy a VM, each vCPU is destroyed along with it. Prior to freeing the VMCS, we remove the mapping from the lookup structure.

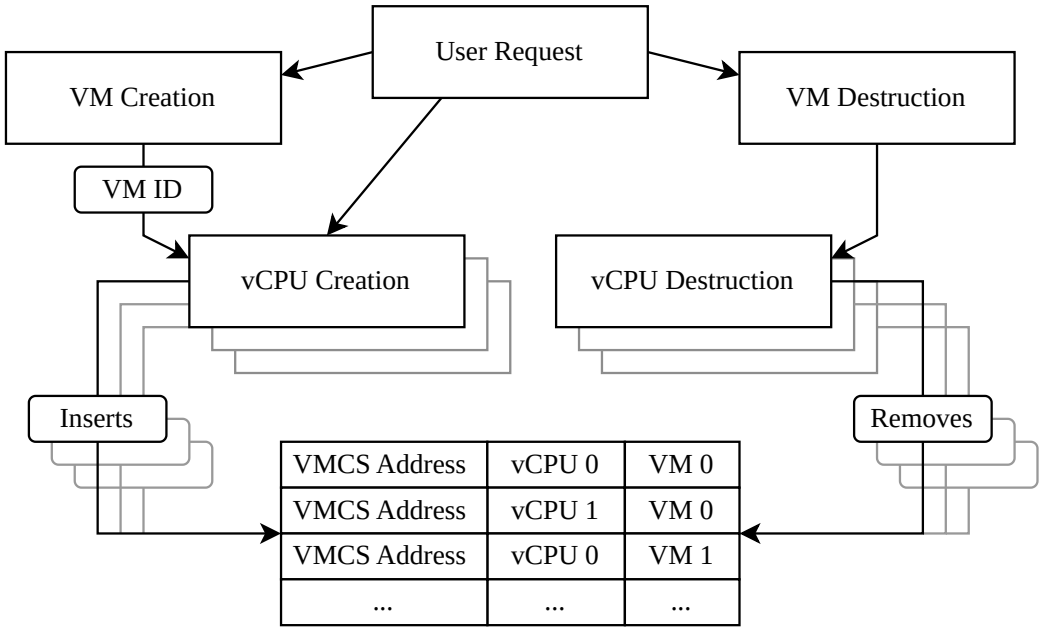

Figure 4.2: VMCS address tracking and mapping

4.2 Process Tracing

We identify processes within a VM instance by their Page Table Address (PTA). Our design tracks the consumed cycles as well as the duration the process runs. Combined with the time the process is loaded, we can attribute energy as described in Section 4.5. We present two different approaches to track cycles and time of processes.

4.3 CR3 Write Exiting

Intel as well as AMD provide hardware features which cause VM exits upon `MOV` to CR3 [14, 15], commonly used for shadow paging. On Intel platforms, the feature is controlled through the VMCS (Section 2.1.1) and can be enabled using the `VMWRITE` instruction.

Once VM exit occurs, hardware writes an exit reason to the VMCS [15, 34] which states whether the exit has been caused by a control register access and can be retrieved by the hypervisor using the `VMREAD` instruction. In that case, we check the exit qualification, also resident within the VMCS, which provides further information about the access type and which control register has been accessed in particular. If we find the cause for the VM exit to be a `MOV` to CR3, we store the PTA. Additional VMI techniques to translate the PTA to a unique

process identifier which is not reused must be applied. This is required as PTAs can be reassigned, if a process terminates and a new one spawns and places its page table at the same address. However, even if no VMI is used, this might not be a problem for most energy diagnostic purposes, as short-lived processes have commonly little to no impact on overall energy consumption.

Furthermore, we retrieve consumed cycles using performance counter and store the TSC value. Upon VM Exit, we read the performance counter and TSC again and calculate the delta of consumed cycles. Finally, we pass the event consisting of the process, start TSC, end TSC, cycle count and the logical cpu on which the exit happened to the *Energy Attribution* component (Section 4.5).

4.4 Intel Processor Trace

Intel-PT generates *Paging Information Packets (PIP)*, which contain the new PTA written in the CR3 register. Furthermore, a non-root bit determines whether the write has been executed on the host or within a guest. Time stamp counter, *Mini Time Clock* and *Cycle Packets* allow estimating the time stamp counter value for each PIP packet, which makes them feasible for start time and duration calculation of processes. The cycle packets are generated with a CYC packet eligible packet type, which includes the PIP packet. However, the frequency of the CYC packet generation can be configured.

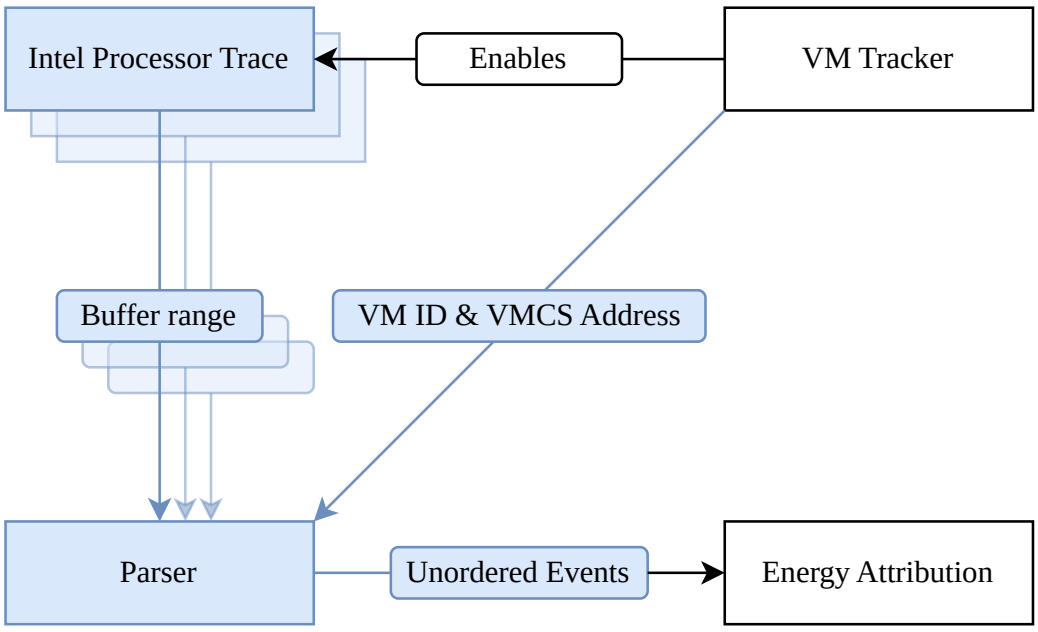

Figure 4.3: Overview of the *Intel-PT* based apporoach

Figure 4.3 shows how the *Intel-PT* approach is integrated into our design. Changes to the basic design overview 4 are highlighted in blue in the figure. In contrast to the *CR3 write exiting* approach, we have to trace each CPU the VM can schedule vCPUs on. Thus, the following has to be done on each logical CPU.

The *Intel-PT* component begins with allocating the trace buffer and filling *ToPA* structure described in Section 2.3.1. It continues setting up the MSRs (Section 2.3.3 and 2.3.1) for *Intel-PT* to produce the required packets and enables it. Furthermore, we register a non-maskable interrupt handler. Our design saves the position within the buffer at which tracing has started. Once an interrupt is handled, we pause tracing, read the producer position and update interrupt and stop positions in the buffer. Placing interrupts is required to avoid buffer overruns and, as a consequence, loosing data. They enable to process smaller parts of the buffer without disabling tracing for an extended period of time. After that, tracing is resumed, and we pass the range [original start position, producer position) to the parser.

When the parser (Section 4.4.4) receives a range from any CPU, it begins parsing the packets. It attributes CPU cycles to each PIP packet and uses the VMCS to identify which VM the CR3 address belongs to. As ranges are received from multiple CPUs, CR3 events parsed might have a later timestamp than events contained in ranges received later on. This results in an unordered stream from a time point of view, thus we have to sort them within the *Energy Attribution* component (Section 4.5).

4.4.1 Configuration

In order to reduce the amount of data generated by *Intel-PT* and minimize the parsing overhead, we only enable the minimum of features required for energy attribution.

As stated in Section 4.4, we require PIP, CYC, MTC and TSC packets to be generated. Therefore, we set `CYCEn`, `MTCEn` and `TSCEn` for the value written to the `IA32_RTIT_CTL` MSR.

In order to reduce the size of the trace, we aim for the largest supported value of `CycTresh`, `MTCFreq` and `PSBFreq`. This poses a tradeoff, leading to a loss of precision in consumed cycles attributed to a process (Section 4.4.4) as well as less granular estimated timestamps per packet. As we only aim to use our approach for board diagnostic purposes, sample with a relatively low frequency and energy consumption provided by RAPL may not be completely accurate. We argue that the resulting effect is negligible. In addition, less PSB packets result in less recovery points in case the trace gets corrupted. As a consequence, bigger parts of the trace become unusable and must be skipped, resulting in more severe data loss.

Furthermore, we rely on ToPA for the buffer, the setup of which we describe in detail in Section 4.4.2. In order to use ToPA, we enable the `ToPA` bit.

As our approach aims to trace everything that runs on a physical CPU, including the guest OS, we set both `OS` and `User` bits (Section 4.4.1).

4.4.2 Trace Buffer

In our design, we configure the *Intel-PT* buffer to behave like a ring buffer using the ToPA mechanism (Section 2.3.1).

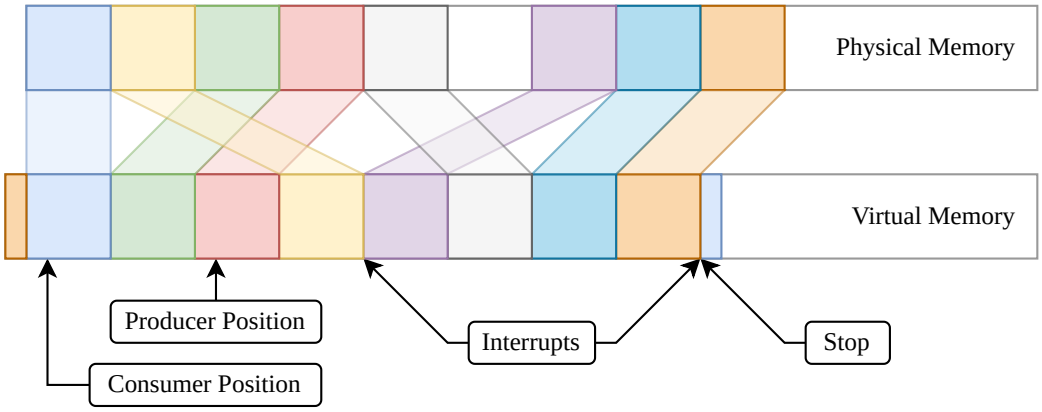

Figure 4.4: Snapshot of the buffer layout before tracing is enabled

Figure 4.4 visualizes the buffer configuration before tracing starts. Each block refers to a physically continuous range of memory with a size of 2^n pages. These blocks are not required to be of the same size, yet we chose to visualize them as such for simplicity reasons. Variable size memory ranges become especially useful if the system is under memory pressure or experiencing high fragmentation and allocating large chunks of physically continuous memory is not possible. The physical offset of each of these blocks are represented as entries within the ToPA table. Their ordering matches the sequence in which data is written to the buffer. The last entry links back to the first one, which ultimately implements a ring buffer mechanism. Once an entry is filled with data, *Intel-PT* continues writing to the subsequent entry unless the filled memory range entry is marked with a `Stop` bit (Section 2.3.1).

In the figure 4.4, the `Producer Position` marks the offset at which *Intel-PT* begins to write data to once `TraceEn` within the `IA32_RTIT_CTL` MSR is set. The `Consumer Position` points to the position which has already been parsed by the Parsing component described in section 4.4.4. It marks the position up to which it is safe to write data to without the risk of overriding unconsumed parts of the trace.

To ease parsing, we map the physically continuous memory ranges to a single continuous virtual memory range. The order matches that of the ToPA entries. Additionally, we map the last physical page of the buffer to the front of the virtual range. Analogously, we map the first physical page to the end. This is indicated by the thin colored boxes in figure 4.4. As a single packet can be split across range boundaries [15], it eliminates the need for special treatment in such cases.

Our approach sets the `Interrupt` and `Stop` positions (Section 2.3.1) using the following algorithm 1 First, the ranges containing the `Producer Position` and

Algorithm 1 Interrupt & Stop bit positioning

Require: $cons\_pos, prod\_pos, buf\_size, num\_ranges$

1: $prod\_idx \leftarrow range\_of(prod\_pos)$
2: $cons\_idx \leftarrow range\_of(cons\_pos)$
3: $int\_cand \leftarrow range\_of((prod\_pos + (buf\_size/4)) \bmod buf\_size)$
4: $int\_cand\_dist \leftarrow (int\_cand - prod\_idx) \bmod num\_ranges$
5: $stop\_dist \leftarrow (cons\_idx - prod\_idx) \bmod num\_ranges$
6: **if** $stop\_dist > 0$ **then**
7: $\quad stop\_dist \leftarrow stop\_dist - 1$
8: $stop\_idx \leftarrow (prod\_idx + stop\_dist) \bmod num\_ranges$
9: **if** $stop\_dist > 0$ **then**
10: $\quad int\_dist \leftarrow stop\_dist - 1$
11: $int\_dist \leftarrow \min(int\_dist, int\_cand\_dist)$
12: $int\_idx \leftarrow (prod\_idx + int\_dist) \bmod num\_ranges$
13: $set\_int\_bit(int\_idx)$
14: $set\_int\_bit(stop\_idx)$
15: $set\_stop\_bit(stop\_idx)$

`Consumer Position` are identified (lines 1-2). The algorithm selects an interrupt candidate positioned up to a quarter of the buffer size forward from the producers current location (lines 3-4). Generally, other interrupt spacing also is possible. In our design, we chose the distance of one quarter so that the parser has enough time to consume the range before data is lost. Algorithm 1 calculates the number of ranges available for writing before the range of the consumer position is reached (line 5). As interrupts and stops are issued at the end of ranges, the index is decremented by one unless they would be placed at the same entry again (line 6-7, 9-10). The stop and an interrupt are added to the range prior to the consumers range (lines 8, 14-15). Finally, the algorithm places an interrupt at least one range before the stop is reached, unless the interrupt candidate is closer (lines 10-13).

In the example figure from 4.4, the interrupt is placed two ranges ahead of the producer position. Furthermore, the stop and second interrupt is set to the range before the consumer, showing the non-conflicting scenario of algorithm 1.

4.4.3 Interrupt Handling

Within the interrupt handler, we first have to check whether the interrupt has been caused by a ToPA event. We achive this by checking the `TraceToPAPMI` bit of `IA32_PERF_GLOBAL_STATUS` [13]. If so, we stop tracing by clearing `TraceEn` from the control MSR. We continue by reading the ToPA MSRs to figure out the position where tracing has been stopped. This allows us to enqueue the range [original start position, stop position) for parsing. `IA32_RTIT_STATUS.Stopped` indicates that we hit the stop within the ToPA entries. If the consumer position is still within the subsequent range, resuming parsing would cause data loss. We postpone re-enabling tracing to until when a slot is free. We lose information about PTA switches up to the point where tracing is enabled again and a PSB packet is generated. As parsing can be done quick (Section 6.4), we only lose track for a small period of time. In order to enable tracing we clear the status MSR, recalculate `Stop` and `Interrupt` positions as seen in Section 4.4.2 and update the ToPA MSRs (Section 2.3.1). Finally, we set `TraceEn` again to resume tracing.

4.4.4 Parsing

The parser receives ranges ready to be parsed from all CPUs with tracing enabled. When `TraceEn` is set, *Intel-PT* generates a PSB packet. While parsing, we first scan for it to ensure following packets are not corrupted or interpreted in the wrong way.

Furthermore, time tracking must be applied as described in the Intel SDM [15] (Estimating TSC within Intel-PT).

Once we decode a VMCS packet, we store the address and use it to identify which VM can be scheduled on the CPU the range is received from. Following PIP packets contain a non-root bit which states whether the PTA switch happened within VMX root operation mode. If it is not set, it means the switch has been caused by the host OS. Otherwise, it has been caused by the guest.

Prior to attributing cycles, the parser has to translate the PTA to a unique process ID as described in Section 4.3. As the parsing happens after a delay, a history of changes must be kept.

We account subsequent CYC packet payloads to this process until a new one occurs. As CYC packets are only generated after a threshold, some cycles might be attributed wrongly as shown in figure 4.5. The colored background refers to

the attributed cycles to a process with the same color. Ranges between CYC packets and process switches highlighted with diagonal hatching denote wrongly attributed cycles. For example, the cycles consumed between the first CYC packet and a process switch belong to the process that was running before the switch but is attributed to the following one.

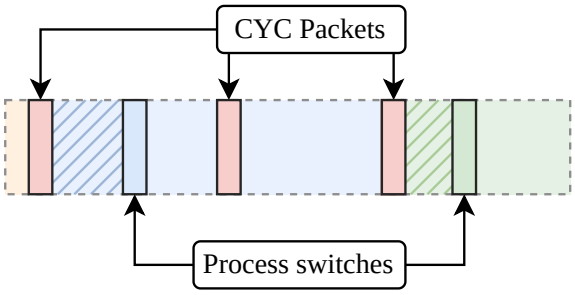

Figure 4.5: Incorrect attribution of cycles to processes

On each PTA switch, we store the timestamp and process ID to determine the duration. We issue an event annotated with the last switch's process ID, the start timestamp, the duration and the estimated cycles while the process has been executing. Once the range has been parsed, we check whether *Intel-PT* has been stopped to avoid data loss. If so, we restart tracing as described in Section 4.4.3.

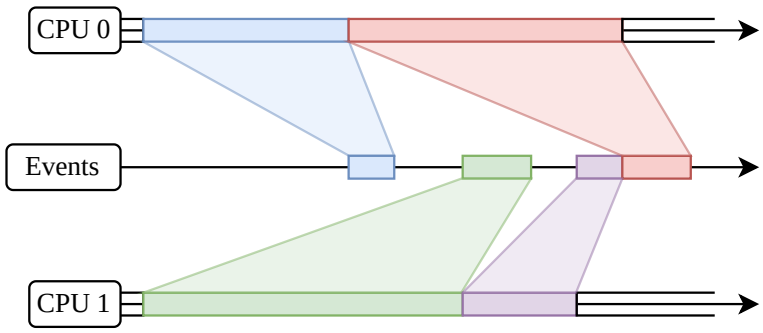

Figure 4.6: Timeline of parsed events

Figure 4.6 shows the stream of parsed events (the stream in the middle) generated by two CPUs and puts it in relation of time (upper and lower stream). The CPU streams refer to the range's (the colored boxes) content in real time. Each point within these ranges correlates to the expected timestamp. The center timeline shows the parsed events contained within these ranges. The position on the scale correlates to the time the event is available to the energy estimation component. It visualizes how parsed events are not strictly ordered. For instance,

events of the red range are parsed after the purple range, thus the timestamps of them are smaller than the ones of the purple ones.

4.5 Energy Attribution

To be able to attribute energy to specific processes, we need to know how much energy the system consumed. We resort to the RAPL interface provided by Intel and AMD (Section 2.2). Furthermore, we use performance counters (Section 2.3.4) to track consumed cycles on each CPU. Both of these statistics are sampled in a fixed interval and stored in slots within the history. We use that information to attribute the energy to processes based on consumed cycles.

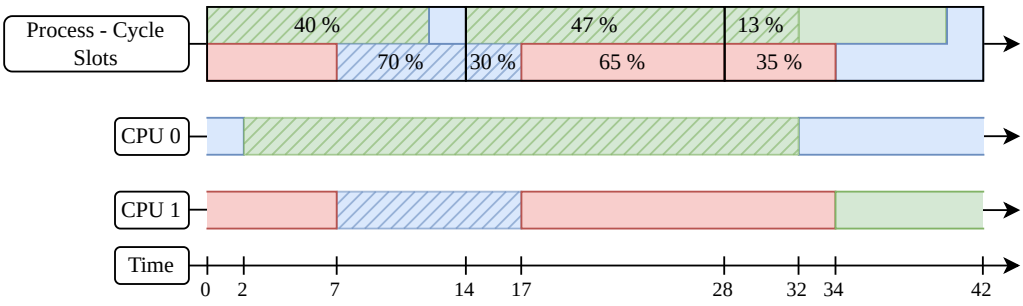

Figure 4.7: Process cycle to slot attribution

Figure 4.7 shows the attribution of consumed cycles by processes to slots within the history. Each colored box refers to one specific process. Rectangles outlined in black visualize a slot within the history for which consumed full system cycles and energy consumption is known. Measured cycles and time are attributed to each slot.

The following equation specifies the cycle to slot attribution. T_s refers to the start time of a slot or process, T_e to the end time and CYC_{proc} expresses the estimated consumed cycles of a process within a slot.

$$CYC_{proc}^{slot} = CYC_{proc} \frac{\max(T_s(slot), T_s(proc)) - \min(T_e(slot), T_e(proc))}{T_e(slot) - T_s(slot)}$$

Cycles are attributed based on the proportion of time the process runs during the slot. We require a lookup data structure for accomplishing this, mapping from process to spent time and cycles.

This attribution approach poses a source of inaccuracy. For example, when a process frame spans multiple slots, the majority of cycles may be consumed in the first slot and very few in the second. The attribution averages this imbalance

across both slots. As a result, the consumed cycles of the whole system may be smaller than the attributed cycles. However, in practice, this is rarely an issue due to the typically short scheduling intervals on most systems.

In figure 4.7, the percentage of processes within the slot timeline refer to the percentage of time the process spends within a slot. For instance, the green process which runs from time point 2 to 32 spends 40 % within the first slot, 47 % in the second and 13 % in the third slot. The cycles consumed by a process are attributed to a slot according to this percentage.

The energy attribution is expressed by the following equation. CYC_{slot} refers to the consumed cycles of the whole system and $Energy_{proc}$ to the energy consumed within a slot.

$$Energy_{proc}^{slot} = Energy_{slot} * \frac{CYC_{proc}}{CYC_{slot}}$$

Thus, inaccuracies caused by cycle attribution propagate to the estimated energy.

As data from *Intel-PT* becomes available after an interrupt happens and the buffer range has been parsed, we have to save quite a bit of history. There is the option to manually stop and restart *Intel-PT* after the end of each interval. However, doing so impacts performance due to smaller buffer sizes (Section 6.4). Based on the last known TSC value from each CPU, slots which do not change anymore can be determined by finding the minimum of those as *Intel-PT* buffer ranges parsed later will have a greater TSC value. In contrast to events received by the *Intel-PT* approach, the time point up to which attributed cycles do not change anymore for *CR3 Write Exiting* is the minimum of the most recent VM Exit across all CPUs which currently execute VMs.

Chapter 5

Implementation

In this Chapter, we first present our implementation of the *Intel-PT* and *CR3 Write Exiting* process tracing following the design in Chapter 4. Our implementation targets the Linux kernel (v6.16) with Fedora patches applied and is integrated into the KVM hypervisor. We expose configuration options for *Intel-PT* via a DebugFS entry for each logical CPU. Enabling both, *Intel-PT* and *CR3 Write Exiting* is also archived with DebugFS [35]. Entries within DebugFS are useful, as they allow us to perform benchmarks with *Process Tracing* selectively without the need to rebuild the kernel. In addition, they mimic requests to enable tracing by the guest.

On initialization of KVM, we allocate cycle performance events for each logical CPU. Similarly, we rely on the power performance event backed by a perf driver to gather energy information. As we only use the package domain, only one event is required.

Additionally, once a VM is created using the `KVM_CREATE_VM` ioctl [36], we increase a static counter to and assign its value for unique VM identification. For vCPU creation, this is not needed as the user needs to specify an integer ID for the `KVM_CREATE_VCPU` ioctl [36] and KVM uses it internally.

5.1 Intel-PT

Our implementation of the *Intel-PT* approach is based on the perf driver [21] shipped with the mainline Linux kernel. Unfortunately, direct use of the mainline driver is not possible as it heavily relies on the auxiliary area of the perf subsystem which is designed to be used by the user space. Furthermore, it lacks customization options for the trace buffer as well as `IA32_RTIT_CTL` options on efficiency cores. Intel's implementation of the driver reads the `CPUID` on the initializing CPU, which is a performance core, thus it reports features like `EventEn`

as missing on efficiency cores. Modifying it to support our requirements of buffer range consumption and parsing in kernel space happened to be quite a challenge.

In the following Sections, metadata directly related to the *Intel-PT* hardware feature is stored in per-CPU variables, unless stated otherwise.

5.1.1 Trace Buffer

Prior to start tracing on a logical CPU, we allocate 32 high order pages, each consisting of 4 MiB of continuous physical memory. We chose to use equal 2^n sized ranges for faster offset calculation as it allows us to use bit masks instead of modulo operations.

A ToPA table is stored in a single 4 KiB page (`struct topa_page` in listing 5.1). Each 4 MiB range becomes an entry within the table (`struct topa_entry` in listing 5.1), and we set the base and size of it as specified by the ToPA definition (Section 2.3.1).

```
struct topa_entry {
  u64 end : 1;
  u64 rsvd0 : 1;
  u64 intr : 1;
  u64 rsvd1 : 1;
  u64 stop : 1;
  u64 rsvd2 : 1;
  u64 size : 4;
  u64 rsvd3 : 2;
  u64 base : 40;
  u64 rsvd4 : 12;
};
struct topa_page {
  struct topa_entry table[507];
  struct topa       topa;
};
```

Listing 5.1: Source Code of the ToPA of the *Intel-PT* Linux Driver

Listing 5.1 shows the ToPA and entry definitions. It is taken from the Linux *Intel-PT* driver and is also present in our implementation. The `struct topa_table` and `struct topa_entry` are designed to reflect the memory layout required by *Intel-PT* hardware. `struct topa` is added to the end of a table and holds metadata information, e.g., the size of all entries.

Finally, a last entry is added which links back to the first one by setting the base to the one stored in the first entry and setting the end bit.

We continue by creating a virtual mapping and place interrupts and stop bits as described in Section 4.4.2.

5.1.2 Configuration

Once the user requests tracing to be enabled on a CPU, we make the trace buffer known to hardware by storing the table address to `IA32_RTIT_OUTPUT_BASE`. We also store, in `IA32_RTIT_OUTPUT_MASK_PTRS`, the entry index and the offset within the entry. We clear the `IA32_RTIT_STATUS` MSR. The `IA32_RTIT_CTL` MSR is configured as described in Section 4.4.1, and its value is saved for use within the interrupt handler.

5.1.3 Interrupt Handling

Linux provides a common vendor specific PMI handler. Whenever it detects the interrupt to be caused by ToPA, it tries to handle it inside the KVM hypervisor. Only if KVM does not handle it, it falls back to the default *Intel-PT* interrupt handler of the perf driver. As our implementation resides within KVM, we chose to implement our handling routines within the preexisting handler. If we handle the interrupt, the preexisting routines will not execute.

Ranges produced by *Intel-PT* between interrupts are passed to the parser by adding work to a workqueue after reading the buffer positions. Each work item consists of:

- An offset within the buffer relative to the start.
- The size of the range.
- A reference to the *Intel-PT* metadata for CPU identification, buffer size, and the virtual mapping.
- The configured MTC frequency.

Otherwise, the implementation follows the design of Section 4.4.3.

5.1.4 Parsing

We resort to workqueues [37] for parsing ranges. Workqueues are a Linux kernel mechanism designed for asynchronous execution of functions. Enqueued tasks are processed by threads from a worker pool bound to the workqueue. Once a worker thread becomes free and the queue of work is not empty, the function bound to the task is executed. Furthermore, it provides SysFS entries to specify on which CPUs worker threads can execute. Our initial goal was to employ this feature in combination with CPU isolation [38]. However, the scheduler only processes work on the first CPU within the isolated CPU mask, thus preventing

parallel execution of tasks. Prior to enabling *Intel-PT*, we allocate an CPU unbound workqueue. Each task within the workqueue refers to an unparsed range of data within the trace buffer.

Furthermore, we maintain a hash map [39] with VMCS addresses as key and a VM ID and VCPU pair as data. Each time a user creates a vCPU with the `KVM_CREATE_VCPU` ioctl, KVM allocates a VMCS. We insert the VMCS address with the corresponding VM ID and vCPU number into the map. Our prototype does not remove VMCS addresses and stores a complete log of all events. In future versions, handling could be implemented by removing them after all ranges of CPUs with tracing enabled have been parsed. With KVM, this at most affects the energy attribution for an *Energy Attribution* interval, as a vCPU associated with the VMCS is only destroyed if the VM is destructed. Reuse of that address is only possible after the vCPU is gone, which leaves at most a time window of the interval length where cycles may be attributed wrongly.

Libipt

For handling tasks within the workqueue and ultimately parsing trace buffers, we rely on a slightly modified version of *libipt* [19] to attribute cycles to PTA switches. Our prototype does not implement PTA replacements and treats the address as a unique process ID. *Libipt* is a library developed by Intel for parsing traces collected with *Intel-PT*. Furthermore, it handles TSC estimation of packets. To do that, it requires a trace with CBR, MTC, TSC and TMA packets, the configured `MTCFreq`, the TSC / clock ratio and the nominal core crystal clock frequency, the latter two are provided by `CPUID`.

libipt provides workarounds for a set of errata.

We enable the ones affecting our test system.

It is written in C and makes almost no use of the C standard library functionality, thus making it a good candidate for in kernel space parsing. It is designed to work on a single continuous data range. For this reason, we require a continuous virtual mapping of the trace buffer and have to split the range to parse in half if a wraparound in the circular trace buffer happens. We use `vmap` to create this mapping.

In the following, we describe our changes to the library to also add cycle to PTA attribution. *Libipt* uses a `struct pt_time` to keep track of timing within the trace. Each time a TSC, MTC, TMA, CBR or CYC packet is encountered, the instance of the struct is altered to reflect the current state. We add a new entry to this structure in order to accumulate the cycles. Each time *libipt* encounters a CYC packet, we increase the value of the field by the packet's payload. When a PIP packet is decoded, we read the field from the timing structure and reset the accumulated cycles.

Additionally, changes have to be applied for wraparound handling. In order to accomplish this, we split the range at the end of the last range within the ToPA table. We parse the first half, stop at the last complete packet, and save the offset. Based on this offset, we calculate the difference to the end of the ToPA entry. For the second half, this difference is subtracted from the start, ensuring packets can be fully parsed. We update the position of the packet parser to the start position and reapply the boundaries. As previously mentioned, this behavior is required, as packets may be split across ToPA entry ranges. Independent parsing of the split apart range is not possible as the decoder needs to know where a packet starts, which requires a PSB packet. Such a PSB packet is unlikely to be the first one, thus parsing the second part would result in data loss.

Handling Work

As multiple processors can enqueue work and also handle it, we rely on synchronization in the form of a per CPU spin lock protected list which restricts execution ordering of tasks generated by a logical CPU. We maintain a wait queue [40, 41] which blocks until the task is at the head of the list. This ensures that ranges generated by *Intel-PT* earlier in time are finished before later ranges are consumed.

We introduce a new work task, indicating that the next range parse should sync to the next PSB before and then reset the parser configuration. It does nothing other than set a flag in a per-CPU state so that the next parsing range knows to search for a PSB first reinitialize the config. This state also contains an instance of a `struct pt_event_decoder` used for parsing. The synchronization request field is initially set to true. Such synchronizing work items are only inserted if the buffer size changes or trace generation is manually stopped (excluding interrupt trace toggles). The reason for this is that, for those events, either the buffer may change or the configuration used for tracing may change. Parsing with an invalid configuration can result in incorrect TSC estimation.

The reason for storing and reusing the parser is that we preserve internal timekeeping metadata and avoid skipping packets due to searches for PSBs. The TSC might be shifted by the duration during which tracing has been disabled to handle the interrupt and reset ToPA flags. However, interrupts are handled very quickly, and we attribute energy based on a 1 second interval, so the inaccuracy is negligible.

Once the parser task is allowed to execute, we retrieve the decoder from the range-CPU state. Synchronization may be requested for several reasons: parsing errors, being the first range to parse, a manual request from outside the parser, or if the buffer's starting position does not match the decoder's saved position. If any of these conditions are met, we set the MTC frequency that tracing has been

configured with. We notify *libipt* about the maximum non-turbo ratio. Furthermore, we set the TSC ART nominator and denominator needed for TSC estimation. As we reinitialize the whole parser configuration, we also need to set the errata workarounds.

When no synchronization is required, we update the bounds within which the parser can operate and continue parsing normally.

In cases where the range wraps around, we split the range and apply the distance to the second part as described in Section 5.1.4.

If the parser decodes a VMCS packet, we look up the address in the hash map (Section 5.1.4) and identify the VM. Following PIP packets belong to this VM if the non-root bit is set. When *libipt* encounters a PIP packet, we notify *Energy Attribution* and provide it with the consumed cycles up to this packet, the PTA and the estimated TSC value.

Once the range has been parsed, we store unattributed cycles for the CPU the range has been generated on. We continue with reporting the completion of parsing with the most recent TSC to the *Energy Attribution* such that it can determine whether it is safe to report slots to user space. The safe time point is determined by the minimum of parsed ranges across introspected logical CPUs. We consume slots up to this TSC as described in Section 5.3.3.

Next, we update the consumer position on the tracing CPU to mark the parsed range as safely writable. Finally, task is removed from the parsing order list, and we wake up tasks waiting in the wait queue.

5.1.5 TSC Offsetting and Scaling

In contrast to the *CR3 Write Exiting* approach, we require TSC offsetting and scaling to be disabled, at least in our prototype, as it is also applied to TSC packets generated by *Intel-PT*. For a fully functioning implementation, the TSC offset and shift must be tracked and applied when the CPU is within a VM. For the *CR3 Write Exiting* approach, this is not required as we only read the TSC within the hypervisor, thus always receiving unmodified values.

5.2 CR3 Write Exiting

We implement the *CR3 Write Exiting* approach based on patches from KVMI [28].

Once the user enables *CR3 Write Exiting*, we notify the *Energy Attribution* component to start tracing cycles and energy. After that, we enable interception on each vCPU which currently exists within KVM. If the vCPU is currently running, we force a VM exit and enable it afterward. In order to enable interception, KVMI's patch reads the current VM execution controls from a bitmap maintained

as a copy (shadow) within KVM. It enables CR3 write exiting using the `VMWRITE` instruction and updates the shadow.

Once a VM exit due to `MOV` to CR3 happens, the value of it is updated within KVMs internal vCPU data structure.

Akin to enabling the feature, the disable path clears the bit within the VMCS as well as the bit in the shadow. It also notifies the *Energy Attribution* component that tracing cycles and energy is no longer needed.

Right before entering the vCPU, we save the current cycle count of the logical CPU the vCPU shall run on as well as the current TSC. Furthermore, we notify the *Energy Attribution* that the logical CPU is executing a VM. This is required to determine the time point prior to which no further changes will occur within the *Energy Attribution* component (Section 4.5).

KVM employs a fast path for VM exits, which never hands CPU control to other processes and immediately reenters the VM. In our implementation, we treat this case as if the VM were executing the whole time. Only after a slow exit is done, we read the TSC and cycle count again. Finally, after the code path handing potential CR3 updates has been executed, we notify the *Energy Attribution* component with:

- VM Entry TSC
- VM Exit TSC
- VM ID and vCPU
- The consumed cycles between entry and exit
- The potentially new PTA

Our prototype uses the PTA as a unique process ID. Furthermore, the *Energy Attribution* takes this as an indicator that the logical CPU is not executing a VM anymore.

5.3 Energy Attribution

As stated in the design 4.5, the energy attribution is based on slots with a fixed interval. We chose an interval length of 1 second as it is short enough to provide meaningful results, yet long enough to reduce the memory footprint of saving the history. For each slot, we store the end TSC, the energy, and a hash map, mapping from a process consisting of VM ID, vCPU number, and PTA to the estimated consumed cycles of that process within the slot. In addition, we store the consumed cycles of each CPU which falls into the slot's interval.

The history of slots is implemented with a circular buffer . As notifications from the *Process Tracer* component might be received in parallel, we require synchronization of the data used for energy attribution. Each hash map within the history is protected by its own spin lock. For all other data, we resort to a `rwlock`, which its spin locks allow either the data to be read by one or more CPU, or to be modified by at most one thread [42, 43].

Prior to tracing consumed cycles and energy, we allocate a performance event for the RAPL package domain and initialize each hash map within the history. Additionally, we initialize a `hrtimer` [44] timer with a callback.

Once tracing is enabled, we store the start TSC of the first slot in a global variable. As energy and cycles only increase and do not reset between performance counter reads, we need to store the previous value in order to calculate the delta. Thus, we also initialize them prior to starting the timer with an expiry time of 1 Second relative to the current time.

5.3.1 Timer Callback

Once the timer reaches its expiry time, the callback is executed.

Our implementation reads the current TSC and grabs the write lock to insert a new slot to the head of the history. The write lock is required as the speculative adding (Section 5.3.2) of cycles to a slot writes an estimated next TSC to the slot. If there is no space within the history, we drop the last entry by updating the tail index and reinitializing the hash map. After that, the write lock is released again.

We read the energy performance counter, calculate the delta and store it in the new slot while holding the write lock. Additionally, the previous energy value is updated to the read value.

In order to get the consumed cycles of all CPUs, we execute a function on all of them which reads the per CPU cycle performance counter and stores it in the history. Similarly, we update the previously read cycle count.

After that, we update the expiry time of the timer to one second in the future.

Finally, if *CR3 Write Exiting* is enabled, we check for complete slots. We find the minimum of the last VM exit TSC across all CPUs which are reported to currently execute a VM. If no VM is executing, the slot's end TSC is used. We consume slots up to this TSC as described in Section 5.3.3.

5.3.2 Process Cycle to Slot Accounting

When we receive an PTA switch event from the *Process Tracer*, we first acquire the read lock. As the *Energy Attribution* gets notified with the new PTA, we have to store the previous running process for each logical CPU. Otherwise, the PTA to which cycles belong cannot be determined. In the following, the process refers

to the previous PTA paired with the VM ID unless stated otherwise. If *Intel-PT* is used, we also store the previous PTA switch TSC and use it as the start TSC.

If the process has no consumed cycles, the TSC range is invalid, or the process end TSC is before the first slot's start TSC, we skip further processing. When the process start TSC is earlier than the first slot's TSC, we proportionally drop cycles and adjust the process start TSC to match the first slot's start TSC. We continue by finding the slot indices of the process start and end TSC using binary search.

If the required slot does not exist as the range extends to a pending timer expiration, we release the read lock, grab the write lock, and recheck the conditions mentioned earlier. We speculatively extend the slot's end TSC by one second past the latest slot if history is not full. Otherwise, we abort and skip accounting for the process. Additionally, we update the process end TSC index to the new slot, release the write lock, and reacquire the read lock.

For each slot in the range, we attribute consumed cycles proportionally based on the duration the process spent in that slot. The following formulas show how the duration is calculated:

$$start = \max(proc\_start\_tsc,\ slot\_start\_tsc)$$
$$end = \min(proc\_end\_tsc,\ range\_end\_tsc)$$
$$duration = end - start$$

The process cycles attributed to this slot is determined by multiplying the process cycles with the duration spent in the slot divided by the total process duration. Under spin lock protection, we try to find the process in the hash map of the current slot. If it exists, the cycles are added. Otherwise, we insert a new entry with the process cycle count for this slot.

Finally, we release the read lock and update the previous process to the incoming PTA and VM ID and vCPU number. In case *Intel-PT* is used, the previous PTA switch TSC is updated.

5.3.3 Consumption of Slots

When a safe time point is determined, we can safely consume slots prior to the time point.

First, we grab the write lock. If we determine there are no slots prior to the safe time point, we release the write lock and return. Otherwise, we iterate through slots up to the slot containing the safe time point.

For each slot, we continue as follows: First, we sum the consumed cycles of each CPU. For each process in the hash map, we report the estimated energy to user space as described in Section 5.4. The estimated energy is calculated by multiplying the energy for the current slot by the cycles consumed by the process divided by the cycles consumed by the whole system.

Finally, we drop consumed slots from the history, update the first slots start TSC to the end TSC of the last slot which has been dropped. Additionally, hash maps are reinitialized and we release the write lock.

5.4 Energy Reporting

We chose to report energy to user space using the relay interface [45]. Once a slot from the *Energy Attribution* component is ready, we write packets consisting of:

- The VM ID as described in Chapter 5.
- The vCPU ID as described in Chapter 5.
- The end TSC of a slot. As slots are based on the same interval (1 Second), a start TSC is not needed. If the user is interested in the exact start TSC, it can be determined by remembering the end TSC of the previous slot.
- The PTA of a process which has been running within this slot.
- The energy attributed to the PTA.

to the global relay channel.

Within user space, one can open the relay file residing in the DebugFS and poll for incoming events. We also used this mechanism for debugging as data of *Intel-PT* and *CR3 Write Exiting* is generated with high frequency.

Chapter 6

Evaluation

In this chapter, we evaluate the energy and performance overhead of *Intel-PT* and analyze the feasibility of *Intel-PT* for consumed energy estimation of in-VM processes. First, we give an overview of packets generated by our test system under various load conditions and discuss the meaning of the amount of data generated. In addition, we analyze the performance and energy overhead *Intel-PT* poses. Furthermore, we evaluate the parsing performance for different buffer sizes. Finally, we evaluate the performance of the *CR3 Write Exiting* approach and compare it to *Intel-PT*.

6.1 Test Setup

Table 6.1 shows the system setup we used for evaluating *Intel-PT* and our implementation.

In Table 6.2 the important *Intel-PT* features supported on the Intel Core i5-14600K are listed. Even though it is stated in the Intel documentation that *Intel-PT* Event Trace has been introduced with Gracemont and Power Event Trace with Goldmont Plus, a predecessor of Gracemont, the efficiency cores report it as unsupported.

CPU errata might also affect our test setup, their actual impact is not completely obvious. We list them for completeness in Table A.1.

6.2 Packet Sizes

In order to determine what an efficient *Intel-PT* configuration should look like, we analyze how often packets of specific types are generated and to which size they accumulate within a time frame. We evaluate consistency of the packet stream using multiple load types. For each of the following traces taken on logical

CPU	Intel Core i5-14600K
Cores	14 (6 P-Cores 12 Threads) & 8 (E-Cores)
Frequency	3.5/2.6 GHz (max Turbo: 5.3/4.0 GHz)
L1	80 KB (P-Core) 96 KB (E-Core)
L2	2 MB (P-Core) / 4 MB (4 E-Core)
Total L2	20 MB
L3	24 MB (shared)
TDP	125 W / 181 W
DRAM	4 × 16 GB Kingston 9965800-015.A00G DDR5-4800
Motherboard	ASUS Pro Q670M-C
CPU Features:	
Hyperthreading	enabled
Turboboost	enabled
Linux	Fedora 42 Server Edition
Kernel Version	v6.16 with Fedora patches applied
Hypervisor	QEMU+Customized KVM
VM Image	Fedora 42 Cloud
Number of vCPUs	4
Assigned Memory	4 GB

Table 6.1: Evaluation system description

CPU 4, we start two VMs and pin the vCPUs to one logical processor such that they overlap as shown in Table 6.3.

We opted to use two VMs as it provokes VMCS switches. It is reasonable as cloud providers may schedule multiple VMs on a logical CPU to increase CPU utilization. Parsing has been configured to happen on CPUs 8-11 so they do not infer with the traced CPU. For each packet the parser sees, we log the type, the size and the estimated TSC to the relay channel. While tracing, we read from the channel and store the results to a file. For each tested configuration, we enable tracing and wait for 60 seconds. After that, we disable tracing. Our implementation either parses after an interrupt happened or parsing has been disabled by a write to the DebugFS entry. Packet statistics are collected by a user space receiver of the relay channel pinned to CPU 0.

As our prototype scans for the first PSB packet if tracing has been manually stopped by a DebugFS write, the first few seconds have been discarded by the parser. Thus, our results show the collected packets 15 seconds after tracing has been enabled for a time frame of 30 seconds.

Feature	P-Core	E-Core
CR3 Filtering	Yes	Yes
Number of Address Ranges	2	2
Configurable PSB and CycleAccurate Mode	Yes	Yes
Valid Configurable PSB Frequency encodings	0-5	0-5
Valid Cycle Threshold encodings	0-5	0-5
MTC Supported	Yes	Yes
Valid MTC Period encodings	0, 3, 6, 9	0, 3, 6, 9
PTWRITE Supported	Yes	Yes
Power Event Trace	No	No
Event Trace	No	No
TNT Disable	No	No
ToPA Output	Yes	Yes
ToPA Tables Allow Multiple Output	Yes	Yes
Single-Range Output	Yes	Yes
IP Payloads are LIP	No	Yes

Table 6.2: Available Intel Processor Trace Features

vCPU	logical CPU
0	2
1	3
2	4
3	5

Table 6.3: vCPU to logical CPU pinning

CYCEn	CycThresh	MTCEn	MTCFreq	TSCEn	BranchEn	PSBFreq
1	5	1	9	1	0	5

Table 6.4: *Intel-PT* configuration with maximum frequency values and branch tracing disabled

CYCEn	CycThresh	MTCEn	MTCFreq	TSCEn	BranchEn	PSBFreq
1	5	1	9	1	1	5

Table 6.5: *Intel-PT* configuration with maximum frequency values and branch tracing enabled

First, we capture a trace of an idling system with no load induced to either the host or the guests.

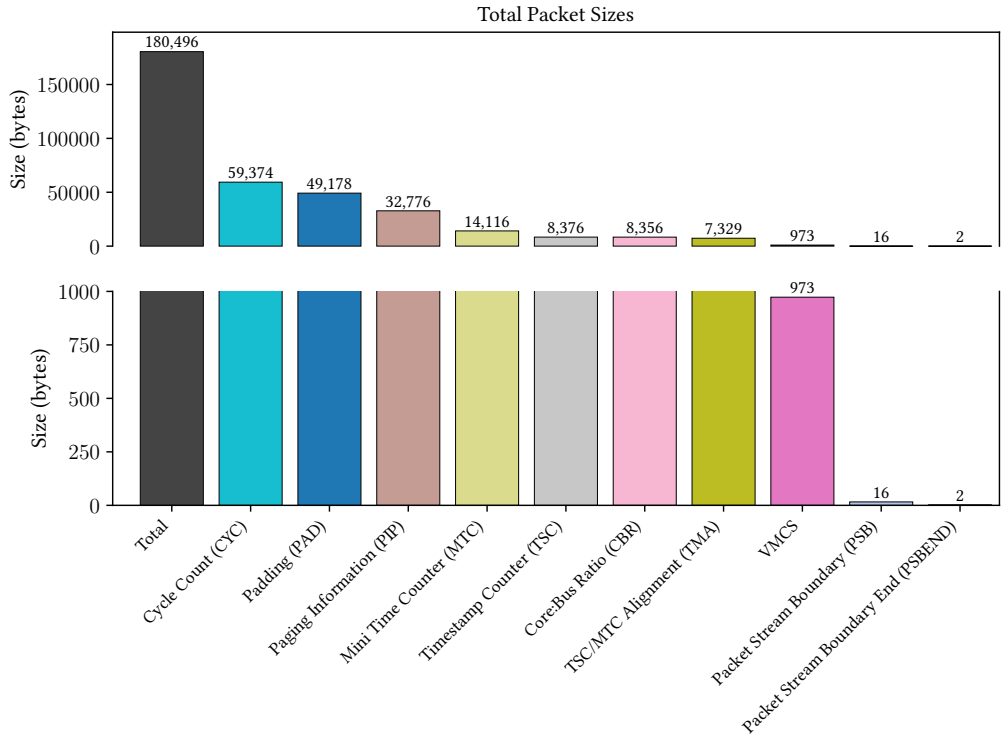

Figure 6.1: Idle packet sizes within the trace buffer with the configuration shown in Table 6.4

Figure 6.1 shows the packet sizes using the configuration shown in Table 6.4 within the time frame 15 s to 30 s past enabling tracing. Even with the highest possible cycle packet threshold, CYC packets take up the most space (33 %) of all packets, followed by padding packets (27 %).

While idling, very few packets, compared to systems under load, are generated.

Akin to Figure 6.1, the results in 6.2 show packet sizes under the same conditions with the config in Table 6.5. CYC packets are still the main size consumer (38 %). TNT packets as well TIP packets are responsible for 49 % of the trace size.

Second, we run a synthetic load on each guest using *stress-ng* [46] (v0.19.05) to consume CPU cycles. Within both VMs, we start *stress-ng* with the `int32` stressor and produce an overall CPU utilization of 20 % on CPU 4 (vCPU 3). The `int32` stressor executes 1000 iterations of a series of bit manipulations (and, xor, shift, …) and arithmetic operations (addition, multiplication, modulo operation,

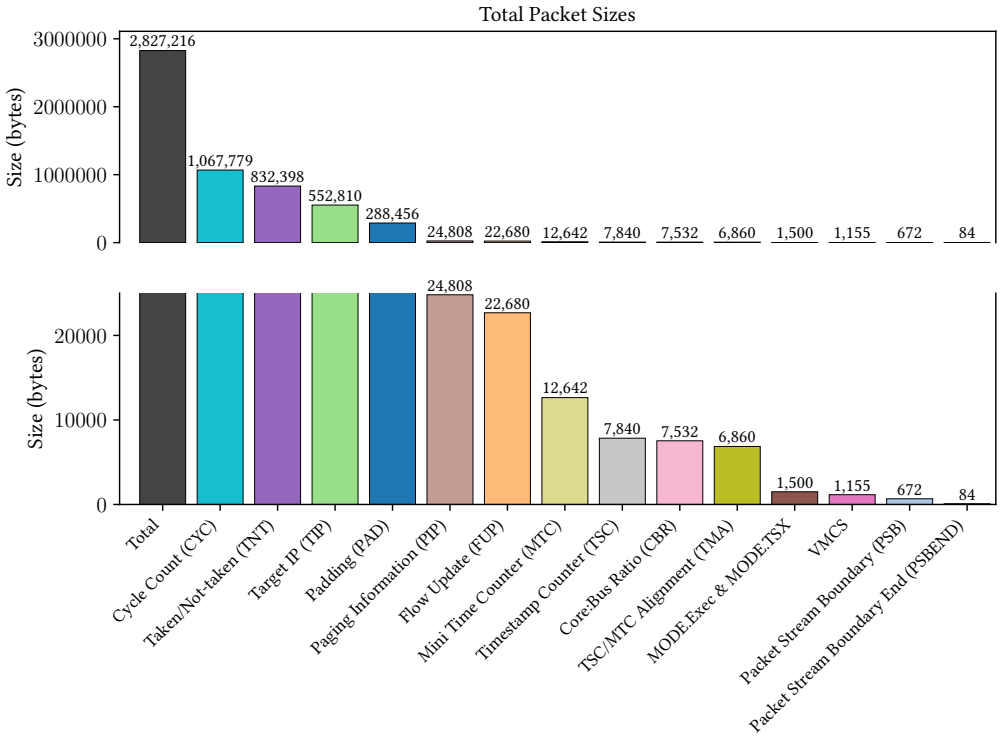

Figure 6.2: Idle packet sizes within the trace buffer with the configuration shown in Table 6.5

...) [47]. *stress-ng* creates the percentage of CPU utilization by sleeping and proceeding executing while tracking time and adjusting durations. Manual validation with the tool `htop` shows that the numbers add up with an occasional margin of 5 %. The benchmark is started with `taskset 0x4 stress-ng --cpu 1 --cpu-method int32 --cpu-load 10`.

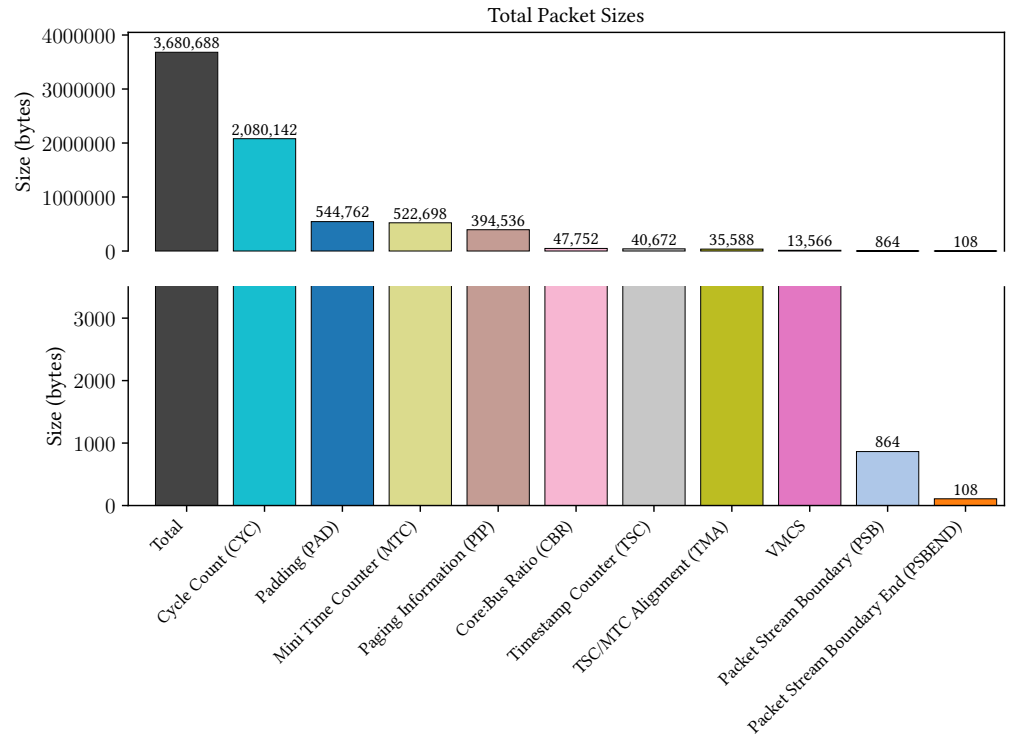

Figure 6.3: Packet sizes within the trace buffer with the configuration shown in Table 6.4 and 20 CPU utilization

In Figure 6.3 we present our results. The CYC packets dominate even more, taking up 57 % of the buffer. Compared to idle load, MTC packets seem to be generated much more frequently.

We found that, when confronted with more than idle load, parsing the buffer results in decoder errors if a PSB is configured to be generated approximately every 2 KB. We are unsure whether it is caused by the hardware generating corrupt packets or by our implementation. Even logging packets to the relay interface with a high frequency may be a reason for the errors.

Thus, even if PSB packets take up only a fraction of the trace, configuring it to the minimal possible value is not feasible and requires further work. When confronted with 100 % CPU utilization, the logged packets show periodically missing

CYCEn	CycThresh	MTCEn	MTCFreq	TSCEn	BranchEn	PSBFreq
1	5	1	9	1	0	1

Table 6.6: Trace Configuration for *Energy Attribution*

slots within the timeline. Due to the lack of time, we did not investigate this issue any further.

As PSB packets are not generated alone and carry metadata packets between PSB and PSBEND, we summed the size of packets within the range, including PSB and PSBEND. We found that for each PSB packet, 86 or 102 bytes of packets are generated, depending on the enabled *Intel-PT* features. For example, if `BranchEn` is set, an additional PAD, 2 $\times$ CYC, MODE, and FUP packets are usually generated. It is to be noted that the Intel documentation makes no statements about when PSB packets are generated, just that the processor makes the best effort to insert them at the requested frequency.

We conclude that, at least for our test system, the configuration in Table 6.6 is sufficient. We choose the `PSBFreq` as small as possible while choosing the largest values for `MTCFreq` and `CycThresh` as both packets take up a lot of space within the trace.

6.3 Analysis of Packet Events in Timelines

In this section, we use the same raw data as in Section 6.4 to create a timeline of packets.

In Figure 6.4 we show a heatmap of padding, PIP, and cycle packets. The heatmap suggests that packets are generated in bursts with idle load. A heatmap of all packets can be found in the appendix (Figure B.1 for idle workload and Figure B.3 with a CPU utilization of 20 %).

6.4 Buffer Size Performance

To evaluate buffer parsing performance, we disable the *Energy Attribution* and only parse the buffer ranges. VMCS addresses are still resolved. In order to make results comparable, we restrict the workqueue affinity to CPU 8 so that the range is only parsed on one preselected CPU. Each time we parse a range, we measure the entry TSC of the parser as well as the exit TSC and log the parsed ranges size. Furthermore, we collect the TSC when an ToPA interrupt happens.

For the benchmarks in this Section we chose the configuration shown in Table 6.6 as it should reflect the energy accounting case. Furthermore, we chose

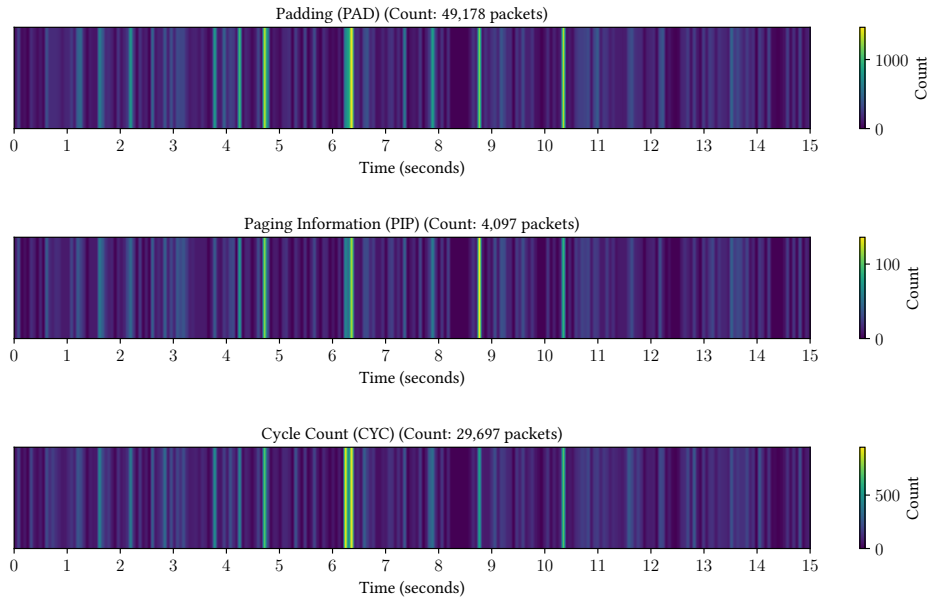

Figure 6.4: Idle packet heatmap with trace configuration shown in Table 6.4

the semaphore *stress-ng* benchmark from Section 6.2 again as PTA switches in combination with time attribution is the main target of interest for energy attribution. The VM setup is equal to the one of Section 6.2. We measure the parsing throughput of our prototype with a variety of buffer constellations. For each buffer configuration, we enable *Intel-PT* for 120 seconds and calculate the averaged throughput using the following formula:

$$Troughput = \frac{sizeof(range\_to\_parse)}{tsc\_to\_s(end\_tsc - start\_tsc)}$$

with tsc_to_s transforming a TSC count to seconds.

In Figure 6.5 and 6.6, a page refers to a high order page, a physically continuous memory range of the size $2^{order} \times PAGE\_SIZE$. We place interrupt position as stated in our design, leading to parsing ranges of one quarter of the whole buffer size.

Figure 6.5 shows our results of CPU 4, a P-core. Parsing larger ranges tend to result in higher throughput.

On the one hand, this can be attributed to the initialization overhead of the parser. On the other hand, we expect that the hardware benefits from predictable memory access as larger ranges result in longer sequences of mostly linear memory access.

We find no performance benefit in parsing ranges larger than 16 MB.

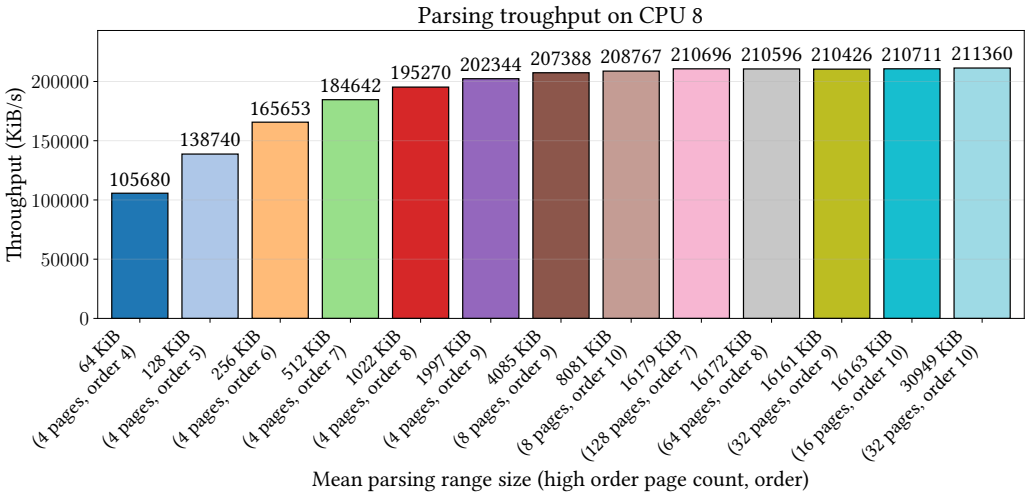

Figure 6.5: Parsing throughput for multiple buffer configurations on P-Core

We argue that this is the case due to longer predictable continuous ranges, thus the CPU can adapt. Additionally, more time has to be spent to handling the interrupts as they occur more often with smaller buffer sizes.

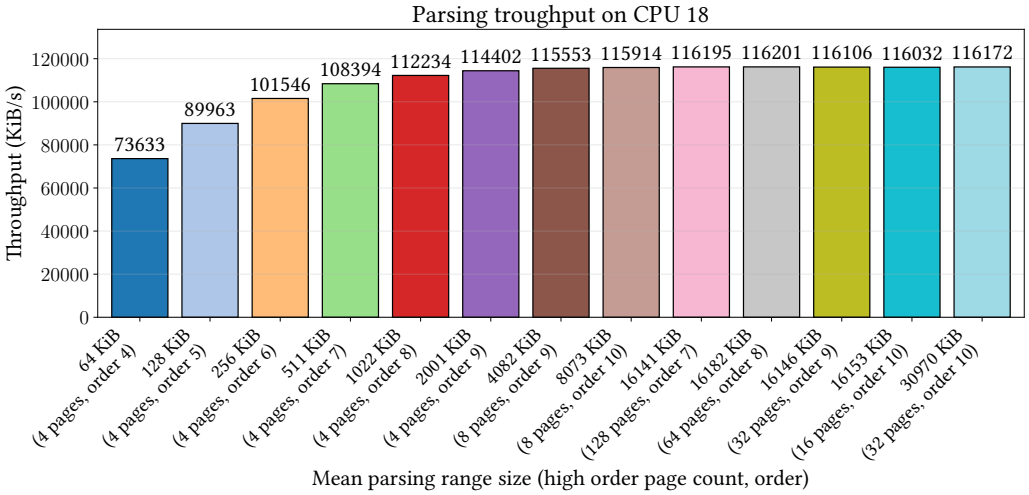

Figure 6.6: Parsing throughput for multiple buffer configurations on E-Core

In Figure 6.6 we present our results for CPU 18, an E-core. Parsing data on P-Cores instead of E-Cores yields a throughput up to twice as high. This effect is particularly observable when parsing large ranges on P-Cores, while smaller ranges are less affected.

For the traces produced by our benchmarks, we find that worst mean interrupt interval to parsing ratio is 1:60 for P-Cores and 1:29 for E-Cores for range sizes of approximately 16 MB.

6.5 Intel-PT Tracing Overhead

In this Section, we evaluate *Intel-PT* for power and performance overhead. At the end of this Section, we provide an overview of the results.

6.5.1 Performance

In this Section, we evaluate the performance overhead of *Intel-PT*. For this, we run benches from the *likwid* [48] benchmark suite on the host machine using `taskset 0x10 likwid-bench -t <bench name> -w S0:1GB:1 -i 5000`. As shown in the command, the benchmarks are executed on CPU 4 and repeated 5000 times. Each benchmark vector has a size of 1 GB.

We evaluate three benchmarks, once with *Intel-PT* enabled but without interrupt set in the ToPA structure, and once with *Intel-PT* disabled:

store_mem which sequencially writes to a vector while bypassing caches.

triad_mem_sse which executes `A[i] = B[i] + C[i] * D[i]` using SSE with bypassing caches.

stream_mem_sse runs `A[i] = B[i] + a * C[i]` using SSE. Similarly, the benchmark bypasses the caches.

All the benchmarks are measured 7 times with the trace configuration we chose for energy attribution 6.6, once with branch tracing disabled and once with it enabled.

Without Branch Tracing

Figure 6.7 shows the memory bandwidth from *likwids* point of view with `BranchEn` cleared. In Figure 6.8, we present the measured execution time. There can be no major difference observed, thus we argue that the slight inaccuracies are the result of background noise.

With Branch Tracing

In Figure 6.9 we present the memory bandwidth results of the three benchmarks with `BranchEn` set. The memory heavy benchmarks `store_mem` and `stream_mem_sse`

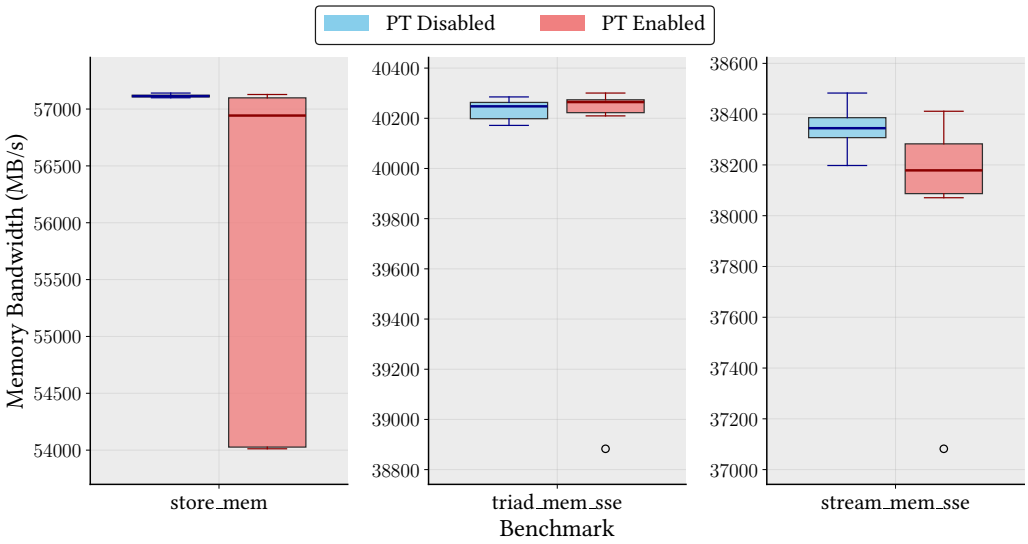

Figure 6.7: Memory bandwidth benchmark results with *Intel-PT* enabled / disabled and branch tracing disabled

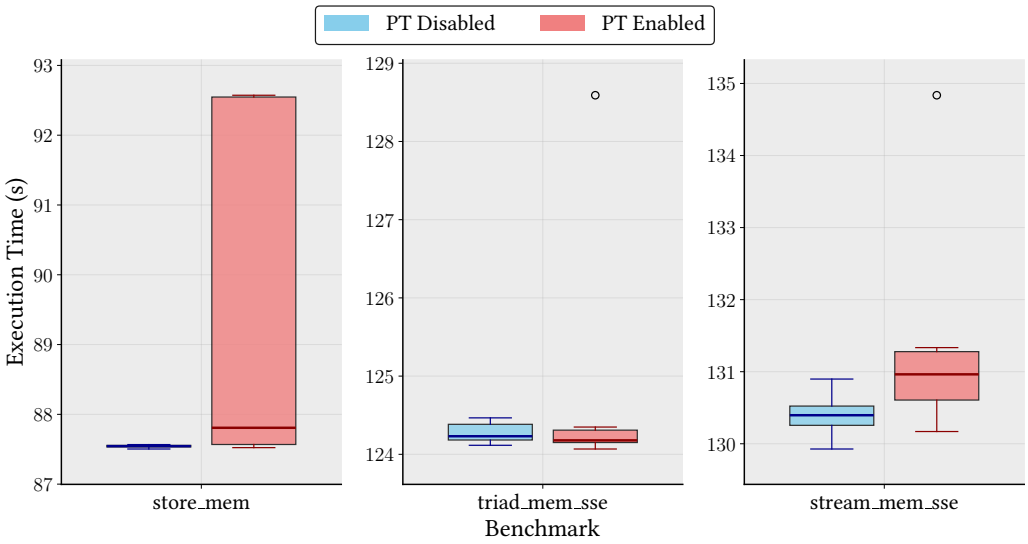

Figure 6.8: Execution time benchmark results with *Intel-PT* enabled / disabled and branch tracing disabled

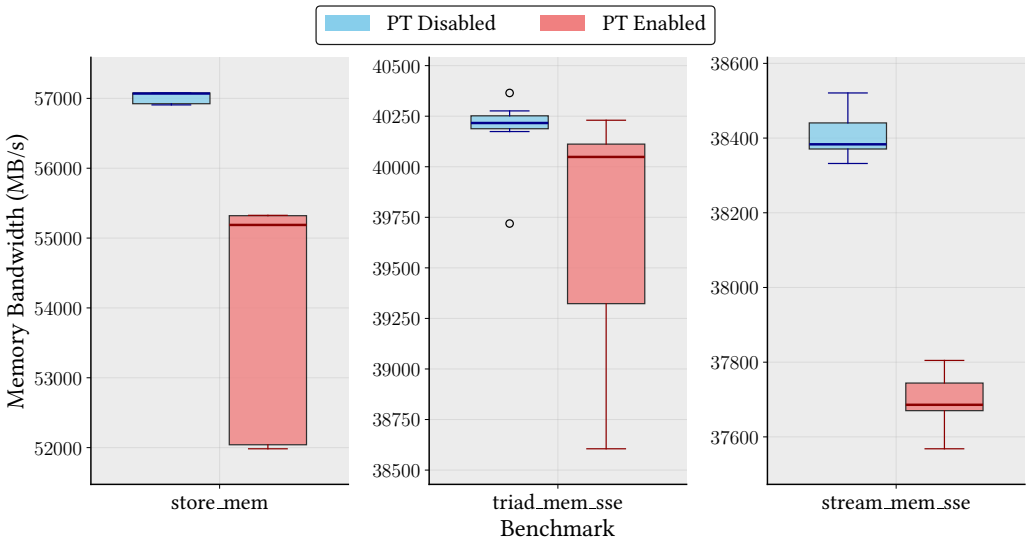

Figure 6.9: Memory bandwidth benchmark comparison with *Intel-PT* enabled / disabled and branch tracing enabled

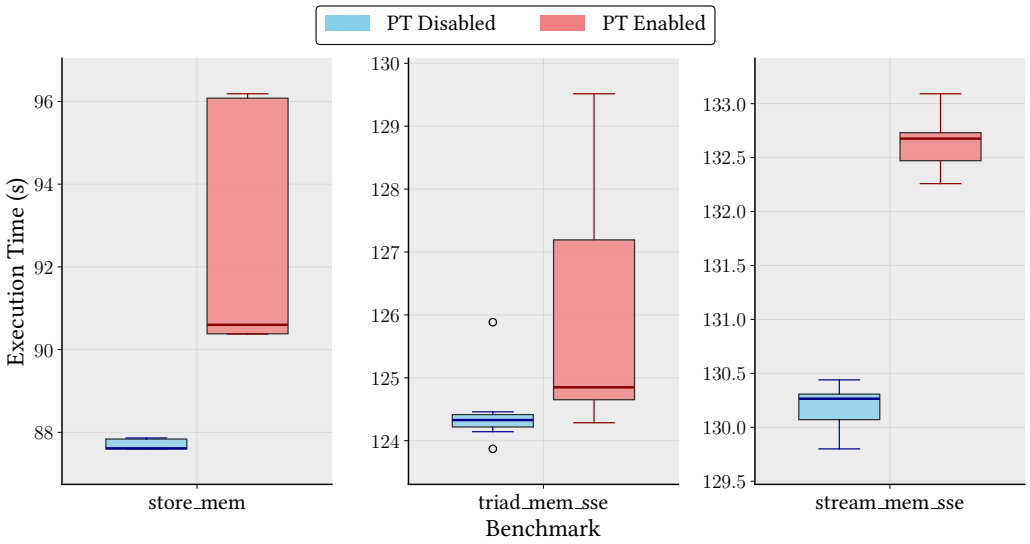

Figure 6.10: Execution time benchmark comparison with *Intel-PT* enabled / disabled and branch tracing disabled

show slight (2 - 8% in the worst case) regressions with *Intel-PT* enabled. In contrast, the `triad_mem_sse` benchmark shows almost no change (0.5 to at most 2 %) in memory bandwidth. We assume this is caused by different access patterns as the triad benchmark accesses four vectors while the stream benchmark touches only three.

Furthermore, Figure 6.10 shows the execution time. Akin to the memory bandwidth measurement, the execution time has also increased for the store and stream benchmark. We suggest that this behavior is induced by hitting memory bandwidth limits, leading to pipeline stalls and overall slowdown.

6.5.2 Power Overhead

Before and after executing the benchmarks of Section 6.5.1, we read the RAPL based energy counter of the Linux kernel driver. By calculating the delta, we get the consumed energy of the system while the benchmark has been executing. For each benchmark run, we calculate $power = \frac{measured\_energy}{execution\_time}$ to avoid misleading results by varying execution times. Not considering the time would result in also measuring the baseline energy consumption of not only the current CPU the measurements have been taken on. It would also include the baseline energy consumption of the whole system.

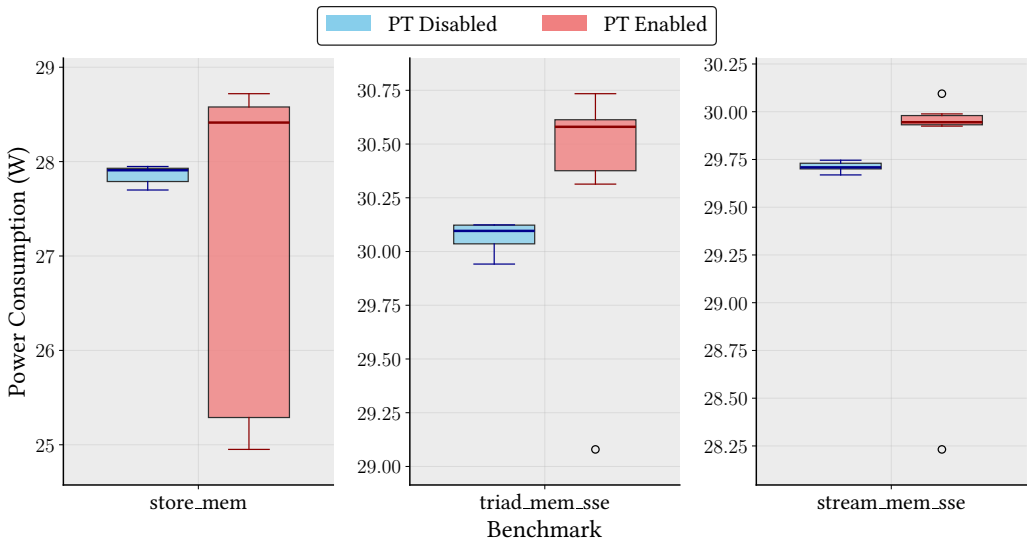

Figure 6.11: Power benchmark comparison with *Intel-PT* enabled / disabled and branch tracing disabled

Akin to the performance while tracing with our configuration, we find a slight difference in our results shown in Figure 6.11 with branch tracing disabled which

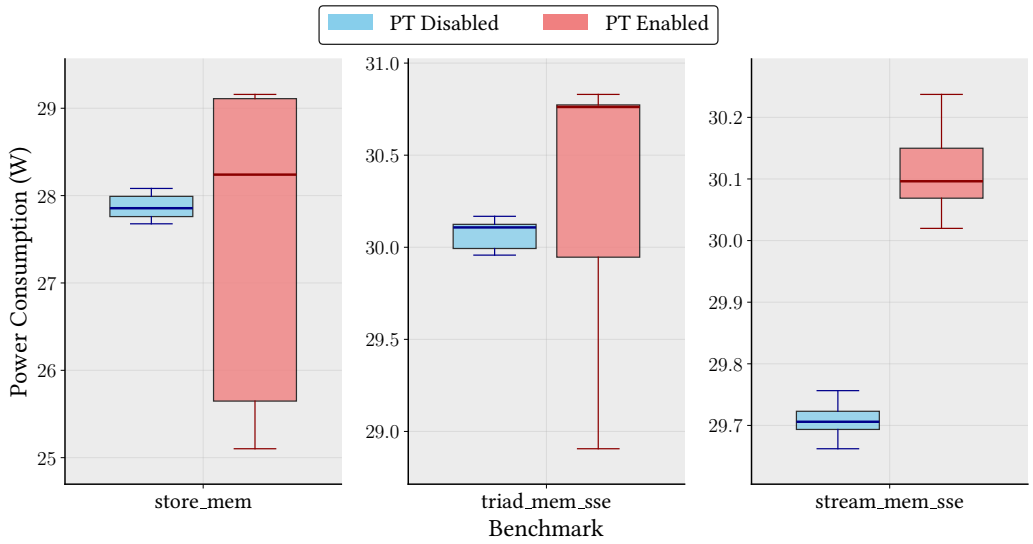

Figure 6.12: Power benchmark comparison with *Intel-PT* enabled / disabled and branch tracing enabled

can be attributed to the energy overhead of *Intel-PT*. We find no major differences in power consumption compared to the results with branch tracing enabled.

Overhead Overview

In Figure 6.13, we visualize the overhead of memory bandwidth, execution time, and power consumption without branch tracing. We find a power overhead of less than 2 % for all three benchmarks. Furthermore, memory and execution time increase by a negligible percentage compared to the baseline with *Intel-PT disabled*.

Figure 6.14 shows the overhead with branch tracing on. Especially for the `store_mem` benchmark, the memory bandwidth and execution time increased by 3 %. There also is an increase in overhead of memory bandwidth and execution time compared to the benchmark with `BranchEn` cleared of 2 % for `store_mem` and `stream_mem_sse`. The power overhead between the benches with and without branch tracing and *Intel-PT* enabled does not change. We assume this is the case and we measure the overall overhead of power consumption and disregard the consumed energy. For the benchmarks, tracing with `BranchEn` set results in longer execution time, which ultimately also increases the consumed energy.

We conclude that the overall *Intel-PT* tracing overhead is negligible for both performance and energy consumption, especially without branch tracing.

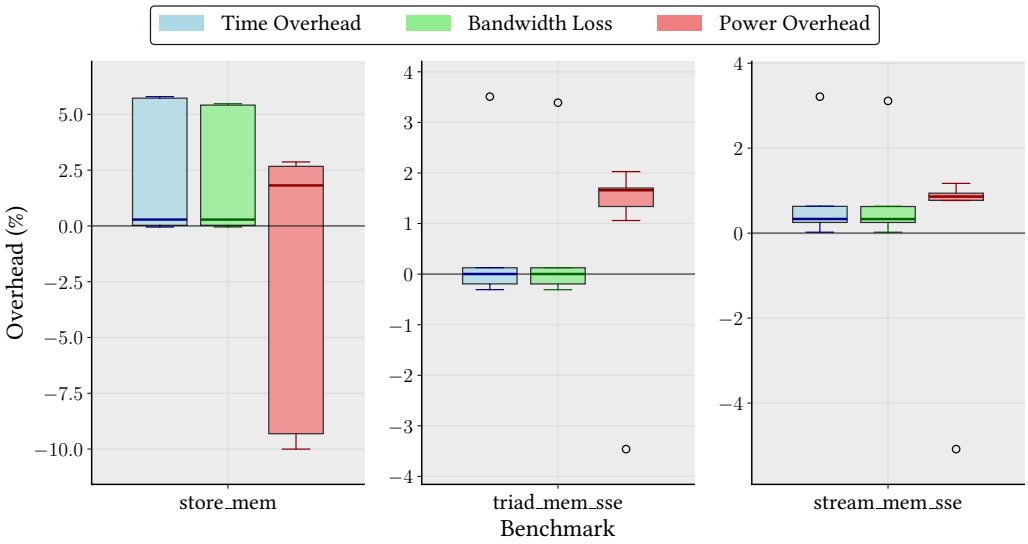

Figure 6.13: Overhead comparison of *likwid* benchmarks with *Intel-PT* enabled / disabled and branch tracing disabled

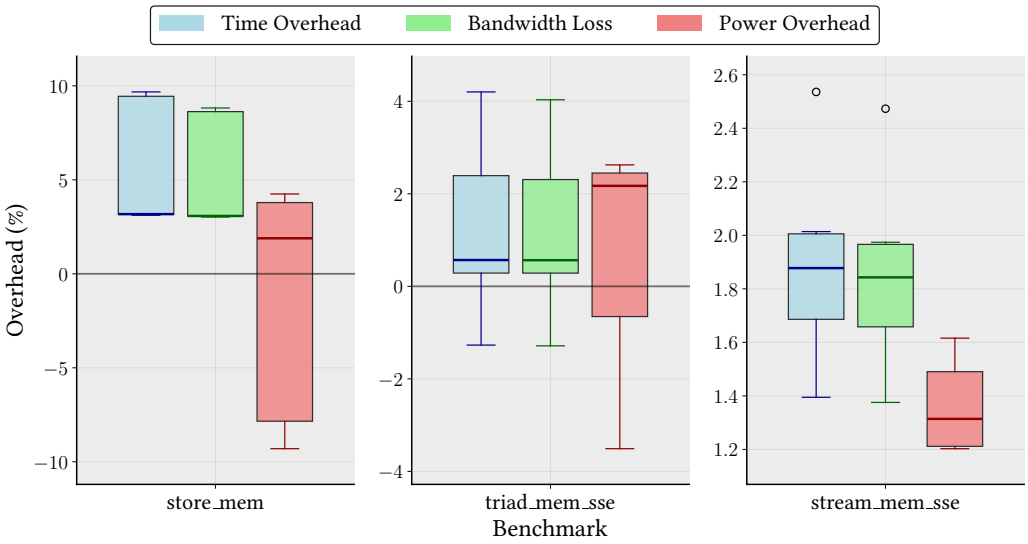

Figure 6.14: Overhead comparison of *likwid* benchmarks with *Intel-PT* enabled / disabled and branch tracing enabled

6.6 CR3 Write Exiting Overhead

In this Section, we analyze the additional cost of VM exits caused by CR3 write exiting. We measure the overhead of *VM exits* and count how often they are caused by CR3 writes. In order to do that, we resort to the perf subsystem which already provides stats about VM exits. It reads KVM trace events issued at VM exits.

We record VM exits with their reasons for various load types induced by *stress-ng* over a time span of 1 minute on vCPU 3. Furthermore, we enable MOV to CR3 write exiting using our prototype. We disable the *Energy Attribution* component such that an VM exit only serves the original purpose.

At the time of recording, there is only one VM started, and the host system is idling.

Following load types are induced:

idle induces no stress. VM Exits seen are caused by interrupts or background noise.

light induce 20 % CPU utilization using the int32 stressor of *stress-ng*. The stressor is described in detail in Section 6.2.

heavy induce 100 % CPU utilization using the *stress-ng* int32 stressor.

switch forces many context switches with the *stress-ng* `switch` stressor. `taskset 0x4 sudo stress-ng --switch 1` creates a parent and child process. The parent process sends a message to the child with a pipe, forcing a context switch [47].

Table 6.7 shows the VM exit reasons collected with `perf kvm stat record -C 4 sleep 60s`. Each row corresponds to a VM exit reason and each column shows the number of exits with that reason collected within the time frame for the load type. The CR Access mean time shows the duration between VM exit and entry of exits caused by CR writes. Empty cells refer to no VM exit of this type within the traced timeframe.

As we only configured VM exits for `MOV` to CR3, CR Access refers to CR3 writes. The switch scenario with artificially induced context switches shows a very high rate of VM exits caused by CR3 writes compared to other load types.

We find that VM exits caused by CR3 writes are not cheap, yet they rarely happen under normal load conditions. If a process forcefully context switches very often, `MOV` to CR3 can degrade performance drastically. Out of the 60 s recording interval, KVM spent 16.5 seconds handling VM exits. This poses an overhead of 27.5 %.

VM Exit Reason	Idle	Light	Heavy	Switch
CR Access	30	26		11149631
MSR Write	2545	53434	60584	60963
MSR Read	59	59		
Timer	51	13681	60015	60016
Interrupt	51	11592	63159	17199
Exception			9	2
`HLT`	822	8463		
`PAUSE`				36
`CPUID`		82		2
EPT misconfiguration				25
Total	3558	87337	183767	11287874
CR Access mean time (ns)	3056	37441		3900

Table 6.7: VM Exit reasons for different load scenarios

If there is only one process running, the VM does not write to the CR3 register at all. Furthermore, MSR write access and exits caused by the timer scales with the load.

The exit count and type heavily depends on the executing process and OS. As `HLT`, `PAUSE` and `CPUID` instructions are available to regular userspace processes, they can intentionally cause exits. Additionally, KVM intercepts MSR reads, writes and CPUID instructions to control available features within VMs.

Even if the VM has been optimized to produce as less VM exits as possible, e.g., by delivering interrupts directly to the VM bypassing the host, an adversary can use such synthetic load to overload our energy attribution approach. Thus, the attribution routines must perform well as they are executed upon every VM exit.

6.7 Discussion

We tested our implementation of the *Energy Attribution* with the *Intel-PT* and *CR3 Write Exiting* process tracers. For that, we put load on vCPUs 0 and 2 in one VM and the same load on vCPUs 1 and 3 of a second one. However, the attributed energy did not match the load. We are still able to attribute the energy to the processes, even if the values are incorrect.

The parsed trace yields both process runtimes and cycle counts, providing detailed execution metrics. Additionally, energy counters can be read at each VM exit, supplementing the trace data with direct energy measurements. We al-

ready do this for cycle count in case the *CR3 Write Exiting* process tracer is used. Existing energy models, such as Kepler (Section 3.2), can use this data as input for energy attribution instead of our implementation. Collecting further performance metrics, such as retired instructions, is also possible and may enhance the precision of such models.

Chapter 7

Future Work

In this chapter, we present additional ideas for the utilization of *Intel-PT* and our *Energy Attribution* design.

7.1 Energy Model Selection

Traditional energy models usually perform well for one specific load type. As shown in the evaluation in Section 6.2, the CYC packets dominate in size. Furthermore, our simple test has shown that the size of the produced data of *Intel-PT* increased by a factor of 20 when switching from an idle state to 20 % CPU utilization. Our evaluation shows that the energy and performance overhead of leaving *Intel-PT* enabled is negligible (Section 6.5.1). We suggest that using the trace data size, e.g., by measuring the interrupt frequency, can be used to classify the load type and, as a result, can further help to select a fitting energy model. However, further research and tests are required to support our idea.

7.2 Continuous Trace Collection with Postponed Analysis

By continuously tracing the system with *Intel-PT* and postponing their analysis, systems can efficiently capture execution behavior without immediate runtime overhead.

If, in addition, no interrupts are set within the ToPA table, not even interrupt overhead occurs. As shown in our evaluation (Section 6.5.1), we found no major performance and energy consumption overhead for a minimal trace configuration, effectively yielding a nearly free, always running debug facility, which is able to look into the past. Furthermore, no VM exits are caused due to tracing.

However, if interrupt bits are not set, parsing the trace might become a problem as a safe start position is unknown. If tracing has been stopped for later analysis, the MSRs contain the current position in the buffer. If there are too few packets generated for a buffer wraparound, the PSB located to the right of the current parser position (assuming tracing writes to the right) may not mark a safe starting point for parsing as it may contain stale data from previous tracings, or even uninitialized memory.

This can even be extended further to trace VMs. Once a VM exit happens, one can swap the trace buffers which only requires a few MSR writes. The trace then can be decoded on a different CPU or socket. If VM exits happen frequently enough, this can be used to attribute the energy of processes without inferring with the CPU the VM runs on, assuming energy metrics can be collected otherwise (in our design this refers to the systems consumed cycle count).

Furthermore, the use of branch tracing might enable attributing energy consumption to functions and help pinpoint what code sections are responsible for high energy consumption. We showed that tracing branches poses a slight performance and energy overhead (Section 6.5.1). However, further work needs to be done for finding a safe start position within a trace buffer and the use of branch packets.

7.3 Page Table Address to Process Identifier Mapping

It remains an open question how PTA addresses can be mapped to unique process IDs and ensuring freshness of data. One apporach is employing VMIFresh (Section 3.1). Furthermore, we suggest to use Intel Page Modification Logging (PML) [15] to identify changes to a previously found kernel memory region, where, e.g., the Linux kernel stores task information in a linked list of `task_structs`. To find this memory region, methods of LibVMI can be used (Section 2.1).

7.4 On Demand Enablement by VMs

Especially in cloud environments where the access to energy metrics and hardware features are restricted, a tool to measure and debug energy consumption is required to optimize applications. We suggest that VMs can request energy consumption reporting based on the approaches presented in our design in Chapter 4.1. This eliminates the need for tracking the energy per process and limits the overhead to the debugging process. To implement such a functionality, we suggest integrating them in existing agents cloud providers ship. An alternative

that does not require installation of agents within the guest system is VMSH (Section 2.1). With the same methods, data can be made available to the guest for consumption.

7.5 Improving Parsing Performance

One of the main bottlenecks of our design is the need of a fast parser. We did not optimize *libipt* for speed. Additionally, there are more parsers for *Intel-PT* available, mainly used by fuzzing systems such as *libxdc* from Schumilo et al. [49] Further work needs to be done to improve the usability of *Intel-PT* for energy attribution.

7.6 Hardware Improvements

As shown in our evaluation (Section 6.2), a large part of the trace consists of padding bytes. They add no additional information and only consume buffer size. If *Intel-PT* hardware emitted less padding packets, the time between interrupts would increase. Combined with our ideas from Section 7.2, we would be able to save longer histories in terms of time.

Chapter 8

Conclusion

In this thesis, we presented two non-invasive VMI techniques providing statistics which can be used for in-VM process energy attribution.

The first, *CR3 write exiting*, causes VM exits on `MOV` to CR3. This allows to trace process starts, and combined with tracing subsequent VM entries and exits, the duration which the process runs. Combined with collecting further metrics such as the consumed cycle count on VM entries and exits, this technique yields enough statistics to attribute energy to processes.

We evaluate the number of VM exits caused by enabling *CR3 write exiting* and measure the duration until reentry with multiple CPU load constellations. We find that VM exits caused by CR3 writes are not cheap, yet they happen rarely enough under normal load conditions to classify it as lightweight.

The second approach employs *Intel Processor Trace (Intel-PT)* and traces the used CPU cycles, the loaded Virtual Machine Control Structure (VMCS) address bound to a vCPU and Page Table Address (PTA) switches. TSC estimation based on *Mini Time Clock*, *Core to Bus ratio*, *Time Stamp Counter* (TSC) and *used CPU cycle* packets allow to assign a TSC value to the PTA switch. This is another method of tracing process start and duration. Additionally, the approximate *consumed CPU cycle* count of a process can be directly parsed from the trace.

We analyze the packet consistency and their trace size with different trace configurations and determine an optimal trace configuration for our test setup and most likely also for future processors. We measure the performance and power overhead of *Intel-PT* with our test configuration designed to be used for energy attribution with both *branch tracing* enabled and disabled. Our findings suggest that tracing with *Intel-PT* poses a minimal power overhead of less than 3 % without branch tracing and less than 5 % with branch tracing. Based on our measurements we suggest that *Intel-PT* poses negligible execution time and memory bandwidth overhead without branch tracing, and can thus be used for energy attribution on process granualrity, even within a VM. With branch

tracing enabled, we observe a mean decrease in memory bandwidth of less than 3 % for our memory-heavy benchmarks. Our measurements indicate that memory bandwidth overhead and execution time increase proportionally for memory intensive workloads.

We propose further research for utilization of *Intel-PT* for energy attribution on function and thread level. In addition, we envision always tracing using *Intel-PT* as the implied overhead is very small. This, in theory, allows saving execution histories, which then can be analyzed, if unusual system conditions, like high and unexpected power/energy consumption is detected, are encountered.

Appendix A

Additional Tables

Erratum	Description	
RPL001	Intel Processor Trace PSB+ Packets May Contain Unexpected Packets	Due to this erratum, FUP and MODE.Exec may be generated unexpectedly.
RPL004	Intel-PT Trace May Drop Second Byte of CYC Packet	A trace decoder may signal a decode error due to the lost trace byte.
RPL017	Intel-PT Trace May Contain Incorrect Data When Configured With Single Range Output Larger Than 4KB	Due to this erratum, a PT trace may contain incorrect values.
RPL025	VM Entry That Clears TraceEn May Generate a FUP	When this erratum occurs, an unexpected FUP may be generated that creates the appearance of an asynchronous event taking place immediately before or during the VM entry.

RPL057	Processor Trace May Generate PSB Packets Too Infrequently	Due to this erratum, trace decoder software may see fewer PSB packets than expected. This may lead to the trace decoder software needing to search further to find a starting point to decode or, when used in circular mode, being unable to decode the trace due to lacking any PSB packets.
RPL058	Processor Trace May Not Generate a CYC Packet Before MODE.EXEC Packets	Due to this erratum, trace decoder software may not be able to precisely determine when mode changes that involve changing the interrupt flag or the application's default operand size happened.

Table A.1: Errata for Intel Core i5-14600K that might affect our tests [50]

Appendix B

Additional Figures

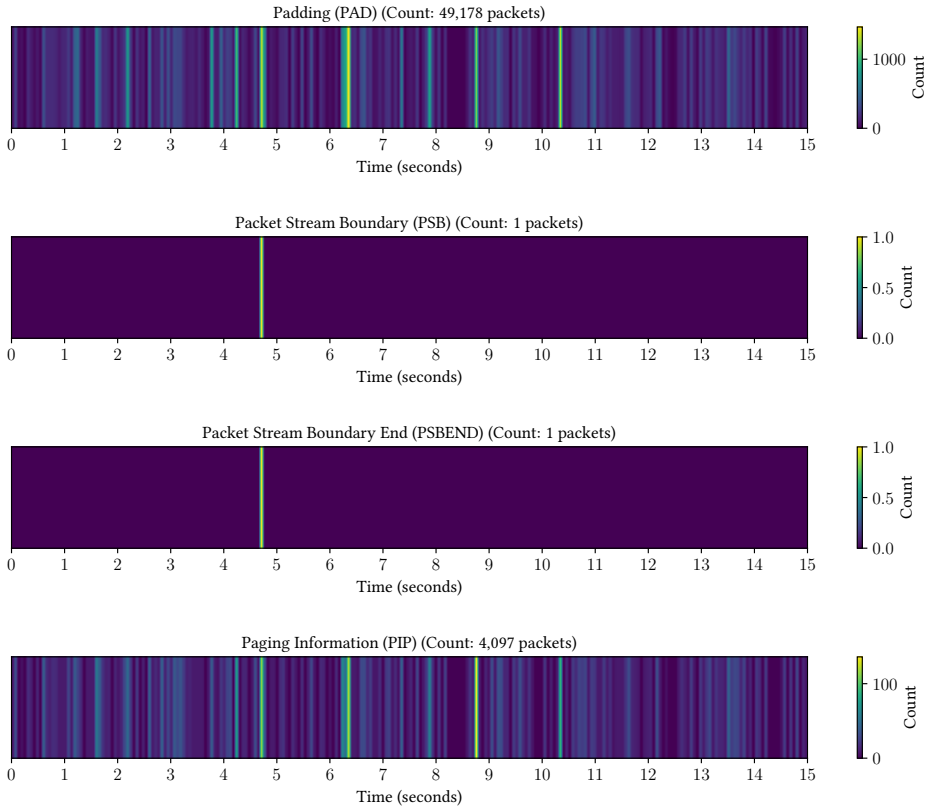

Figure B.1: Complete idle packet heatmap with trace configuration shown in Table 6.4. Created with the same raw data as used in Section 6.2

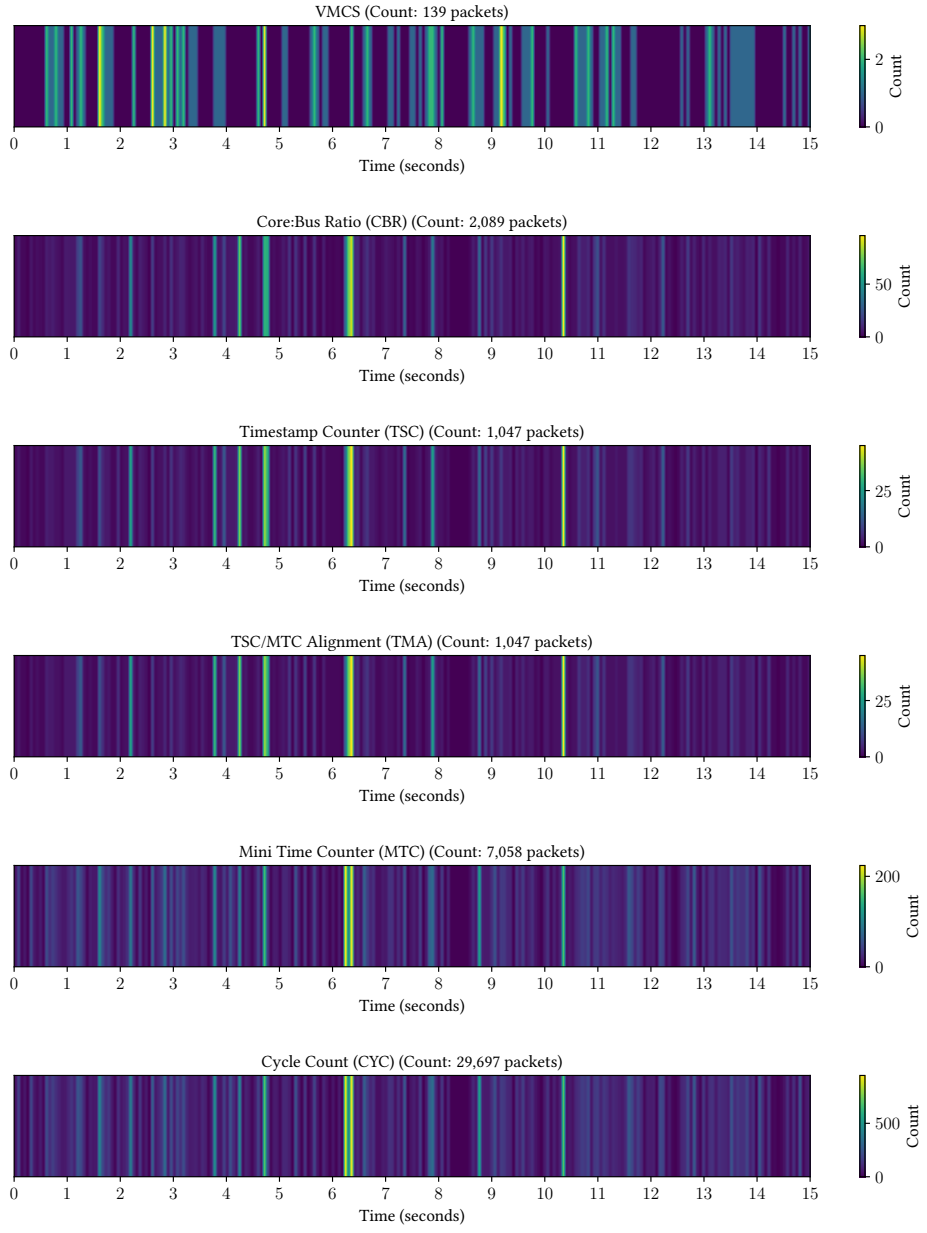

Figure B.2: Complete idle packet heatmap with trace configuration shown in Table 6.4. Created with the same raw data as used in Section 6.2 (continued)

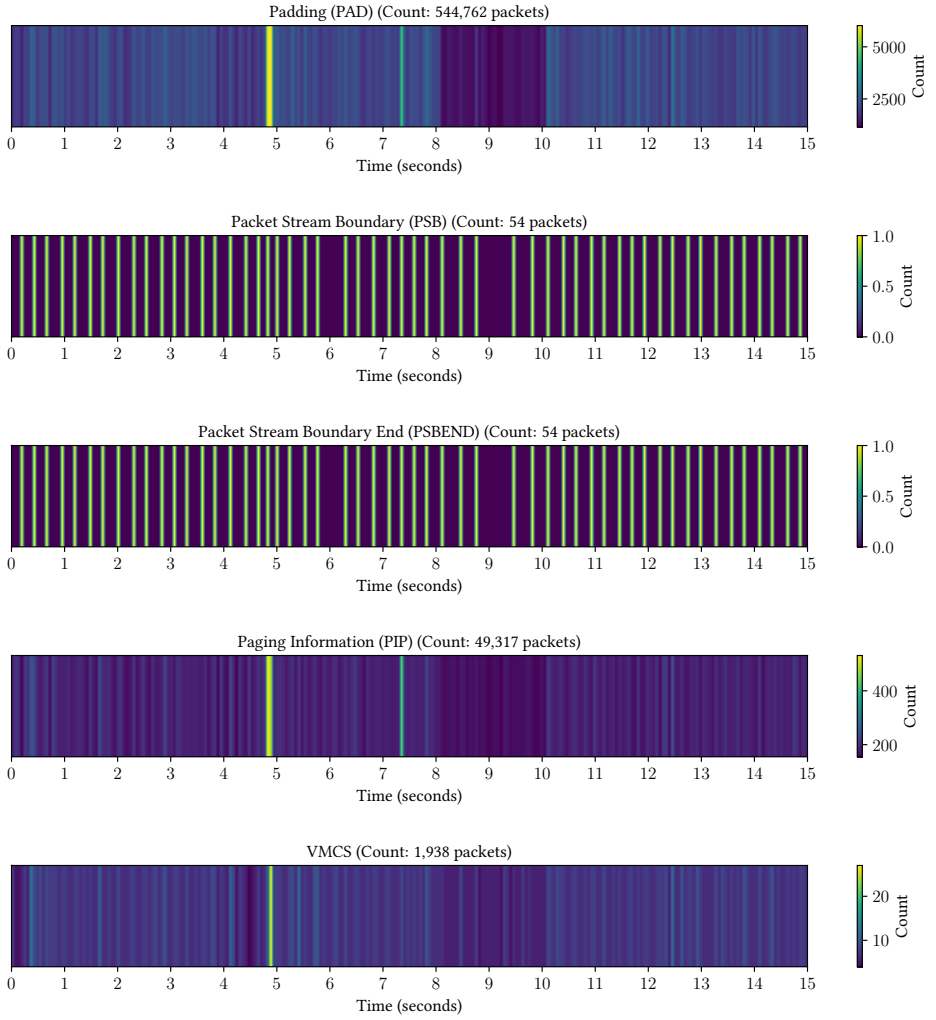

Figure B.3: Complete packet heatmap with trace configuration shown in Table 6.4 and 20 CPU utilization. Created with the same raw data as used in Section 6.2

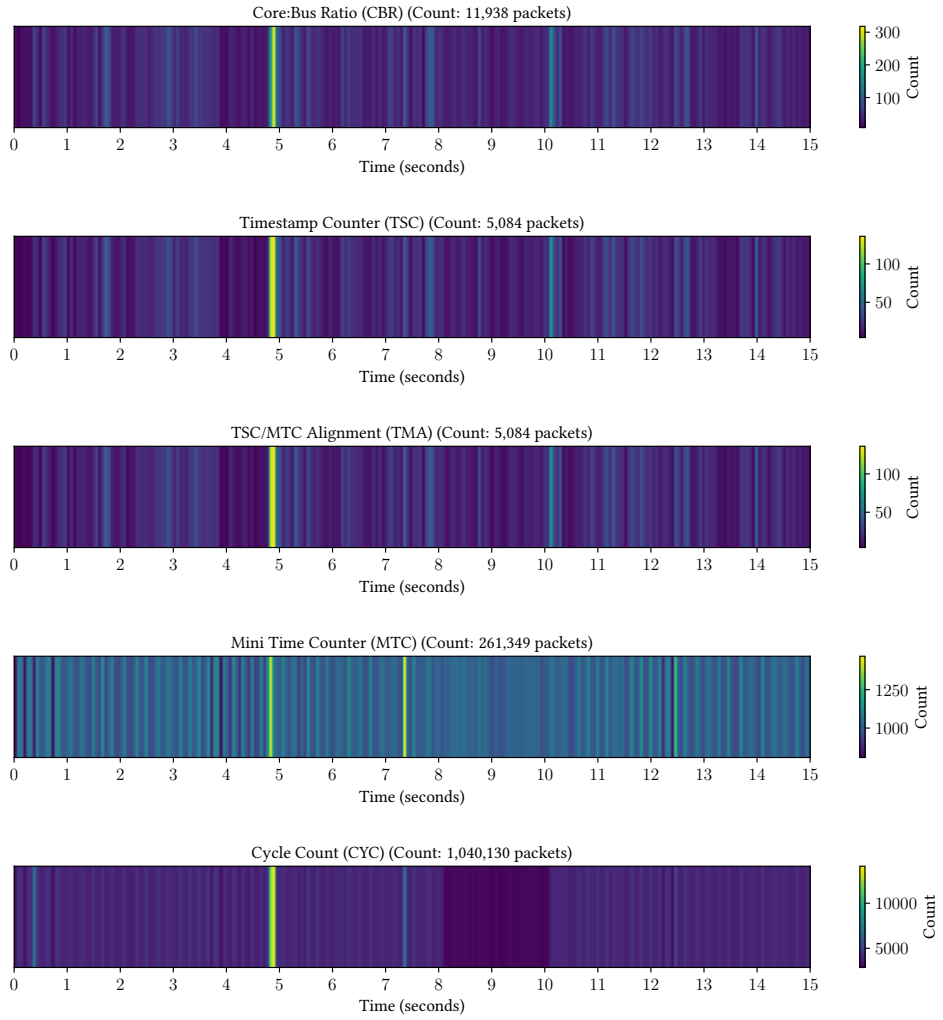

Figure B.4: Complete packet heatmap with trace configuration shown in Table 6.4 and 20 CPU utilization. Created with the same raw data as used in Section 6.2 (continued)

Bibliography

[1] M. Amaral *et al.*, "Process-Based Efficient Power Level Exporter," in *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, ISSN: 2159-6190, 2024-07, pp. 456–467. DOI: 10.1109/CLOUD62652.2024.00058. [Online]. Available: https://ieeexplore.ieee.org/document/10643925 (visited on 2025-04-08).

[2] *Carbon Footprint reporting methodology | Carbon Footprint Documentation*, en, 2025-05. [Online]. Available: https://cloud.google.com/carbon-footprint/docs/methodology (visited on 2025-05-18).

[3] *Viewing your carbon footprint - AWS Billing*, 2025-04. [Online]. Available: https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/what-is-ccft.html (visited on 2025-05-18).

[4] *What is Azure carbon optimization - Azure Carbon Optimization*, en-us, 2025-03. [Online]. Available: https://learn.microsoft.com/en-us/azure/carbon-optimization/overview (visited on 2025-05-18).

[5] M. Lipp *et al.*, "PLATYPUS: Software-based Power Side-Channel Attacks on x86," in *2021 IEEE Symposium on Security and Privacy (SP)*, ISSN: 2375-1207, 2021-05, pp. 355–371. DOI: 10.1109/SP40001.2021.00063. [Online]. Available: https://ieeexplore.ieee.org/document/9519416 (visited on 2025-04-08).

[6] Z. Zhang, S. Liang, F. Yao, and X. Gao, "Red Alert for Power Leakage: Exploiting Intel RAPL-Induced Side Channels," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '21, New York, NY, USA: Association for Computing Machinery, 2021-06, pp. 162–175, ISBN: 978-1-4503-8287-8. DOI: 10.1145/3433210.3437517. [Online]. Available: https://dl.acm.org/doi/10.1145/3433210.3437517 (visited on 2025-05-18).

[7] H. Hè, M. Friedman, and T. Rekatsinas, "EnergAt: Fine-Grained Energy Attribution for Multi-Tenancy," *SIGENERGY Energy Inform. Rev.*, vol. 4, no. 3, pp. 18–25, 2024-09. DOI: 10.1145/3698365.3698369. [Online]. Available:

https://dl.acm.org/doi/10.1145/3698365.3698369 (visited on 2025-04-06).

[8] *VMware ESXi 8.0 Update 1 Release Notes*. [Online]. Available: https://techdocs.broadcom.com/us/en/vmware-cis/vsphere/vsphere/8-0/release-notes/esxi-update-and-patch-release-notes/vsphere-esxi-801-release-notes.html (visited on 2025-05-29).

[9] M. Amaral *et al.*, "Kepler: A Framework to Calculate the Energy Consumption of Containerized Applications," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, ISSN: 2159-6190, 2023-07, pp. 69–71. DOI: 10.1109/CLOUD60044.2023.00017. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10254956 (visited on 2025-04-16).

[10] J. Thalheim, P. Okelmann, H. Unnibhavi, R. Gouicem, and P. Bhatotia, "VMSH: Hypervisor-agnostic guest overlays for VMs," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22, New York, NY, USA: Association for Computing Machinery, 2022-03, pp. 678–696, ISBN: 978-1-4503-9162-7. DOI: 10.1145/3492321.3519589. [Online]. Available: https://dl.acm.org/doi/10.1145/3492321.3519589 (visited on 2025-03-31).

[11] S. Sentanoe, "VMIaaS: Virtual Machine Introspection as a Service," PhD Thesis, Universität Passau, 2024. [Online]. Available: https://opus4.kobv.de/opus4-uni-passau/files/1502/Dissertation_Sentanoe.pdf (visited on 2025-05-20).

[12] *Compute Engine overview*, en. [Online]. Available: https://docs.cloud.google.com/compute/docs/overview (visited on 2025-10-24).

[13] "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2," en, vol. 3B, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[14] "AMD64 Architecture Programmer's Manual, Volumes 1-5," en, 2023-06.

[15] "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3," en, vol. 3C, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[16] H. Xiong, Z. Liu, W. Xu, and S. Jiao, "Libvmi: A Library for Bridging the Semantic Gap between Guest OS and VMM," in *2012 IEEE 12th International Conference on Computer and Information Technology*, 2012-10, pp. 549–556. DOI: 10.1109/CIT.2012.119. [Online]. Available: https://ieeexplore.ieee.org/document/6391957 (visited on 2025-05-17).

[17] Y. Hebbal, S. Laniepce, and J.-M. Menaud, "Virtual Machine Introspection: Techniques and Applications," in *2015 10th International Conference on Availability, Reliability and Security*, 2015-08, pp. 676–685. DOI: 10.1109/ARES.2015.43. [Online]. Available: https://ieeexplore.ieee.org/document/7299979/ (visited on 2025-05-17).

[18] *Overview of System-Wide Power Profile • uProf User Guide 5.0 • Reader • AMD Technical Information Portal*. [Online]. Available: https://docs.amd.com/r/en-US/57368-uProf-user-guide/Overview-of-System-Wide-Power-Profile (visited on 2025-05-28).

[19] *Libipt*, 2025-05. [Online]. Available: https://github.com/intel/libipt (visited on 2025-05-25).

[20] A. Kleen, *Simple-pt*, 2025-05. [Online]. Available: https://github.com/andikleen/simple-pt (visited on 2025-05-25).

[21] *Perf-intel-pt(1) - Linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man1/perf-intel-pt.1.html (visited on 2025-04-17).

[22] *Perf: Linux profiling with performance counters*. [Online]. Available: https://perfwiki.github.io/main/tutorial/ (visited on 2025-10-14).

[23] *Rust-vmm*, en. [Online]. Available: https://github.com/rust-vmm (visited on 2025-05-19).

[24] A. Florescu, *Building the virtualization stack of the future with rust-vmm | Opensource.com*, en, 2019-03. [Online]. Available: https://opensource.com/article/19/3/rust-virtual-machine (visited on 2025-05-19).

[25] *Libvmi*. [Online]. Available: https://libvmi.com/ (visited on 2025-05-19).

[26] B. Payne, "Simplifying virtual machine introspection using LibVMI.," Albuquerque, NM, and Livermore, CA (United States): Office of Scientific and Technical Information (OSTI), 2012-09. DOI: 10.2172/1055635. [Online]. Available: https://www.osti.gov/servlets/purl/1055635/ (visited on 2025-05-17).

[27] *KVMi subsystem for KVM — KVM-VMI 0.1 documentation*. [Online]. Available: https://kvm-vmi.github.io/kvm-vmi/master/kvmi.html (visited on 2025-05-20).

[28] *[PATCH v12 00/77] VM introspection - Adalbert Lazăr*. [Online]. Available: https://lore.kernel.org/kvm/20211006173113.26445-1-alazar@bitdefender.com/ (visited on 2025-05-19).

[29] *The Volatility Foundation - Promoting Accessible Memory Analysis Tools Within the Memory Forensics Community*, en-US. [Online]. Available: https://volatilityfoundation.org/ (visited on 2025-05-20).

[30] T. Dangl, S. Sentanoe, and H. P. Reiser, "VMIFresh: Efficient and fresh caches for virtual machine introspection," *Computers & Security*, vol. 135, p. 103 527, 2023-12, ISSN: 0167-4048. DOI: 10.1016/j.cose.2023.103527. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404823004376 (visited on 2025-04-26).

[31] T. Dangl, S. Sentanoe, and H. P. Reiser, "Active and passive virtual machine introspection on AMD and ARM processors," en-US, *Journal of Systems Architecture*, vol. 149, p. 103 101, 2024-04, ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2024.103101. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762124000389 (visited on 2025-03-13).

[32] C. P. Lamprakos, D. S. Bouras, F. Catthoor, and D. Soudris, "Reliable Basic Block Energy Accounting," en, in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, C. Silvano, C. Pilato, and M. Reichenbach, Eds., Cham: Springer Nature Switzerland, 2023, pp. 193–208, ISBN: 978-3-031-46077-7. DOI: 10.1007/978-3-031-46077-7_13.

[33] M. Amaral, S. Choochotkaew, E. Kyung Lee, H. Chen, and T. Eilam, *Exploring Kepler's potentials: Unveiling cloud application power consumption*, 2023-10. [Online]. Available: https://www.cncf.io/blog/2023/10/11/exploring-keplers-potentials-unveiling-cloud-application-power-consumption/.

[34] "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4," en, vol. 3D, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[35] *DebugFS — The Linux Kernel documentation*. [Online]. Available: https://docs.kernel.org/filesystems/debugfs.html (visited on 2025-10-24).

[36] *The Definitive KVM (Kernel-based Virtual Machine) API Documentation — The Linux Kernel documentation*. [Online]. Available: https://docs.kernel.org/virt/kvm/api.html (visited on 2025-10-19).

[37] *Workqueue — The Linux Kernel documentation*. [Online]. Available: https://docs.kernel.org/core-api/workqueue.html (visited on 2025-10-19).

[38] *The kernel's command-line parameters — The Linux Kernel documentation.* [Online]. Available: https://docs.kernel.org/admin-guide/kernel-parameters.html (visited on 2025-10-19).

[39] N. Brown, "The rhashtable documentation I wanted to read," en-US, *LWN.net*, 2018-04. [Online]. Available: https://lwn.net/Articles/751374/ (visited on 2025-10-19).

[40] *Unreliable Guide To Hacking The Linux Kernel — The Linux Kernel documentation.* [Online]. Available: https://docs.kernel.org/kernel-hacking/hacking.html (visited on 2025-10-19).

[41] *Character device drivers — The Linux Kernel documentation.* [Online]. Available: https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html (visited on 2025-10-19).

[42] *Lock types and their rules — The Linux Kernel documentation.* [Online]. Available: https://docs.kernel.org/locking/locktypes.html#rwlock-t (visited on 2025-10-20).

[43] *Locking lessons — The Linux Kernel documentation.* [Online]. Available: https://docs.kernel.org/locking/spinlocks.html (visited on 2025-10-20).

[44] *High resolution timers and dynamic ticks design notes — The Linux Kernel documentation.* [Online]. Available: https://docs.kernel.org/timers/highres.html (visited on 2025-10-20).

[45] *Relay interface (formerly relayfs) — The Linux Kernel documentation.* [Online]. Available: https://docs.kernel.org/filesystems/relay.html (visited on 2025-10-19).

[46] C. I. King, *ColinIanKing/stress-ng*, 2025-06. [Online]. Available: https://github.com/ColinIanKing/stress-ng (visited on 2025-06-17).

[47] *Ubuntu Manpage: Stress-ng - a tool to load and stress a computer system.* [Online]. Available: https://manpages.ubuntu.com/manpages/focal/man1/stress-ng.1.html (visited on 2025-10-24).

[48] T. Gruber, M. Panzlaff, J. Eitzinger, G. Hager, and G. Wellein, *LIKWID*, 2024-12. DOI: 10.5281/ZENODO.4275676. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.4275676 (visited on 2025-10-24).

[49] *Nyx-fuzz/libxdc*, 2025-09. [Online]. Available: https://github.com/nyx-fuzz/libxdc (visited on 2025-10-24).

[50] "13th Generation Intel® Core™, Intel® Core™ 14th Generation, Intel® Core™ Processor (Series 1) and (Series 2), and Intel® Xeon™ E 2400 Processor, and Intel® Xeon™ 6300 Processor Specification Update," en, [Online]. Available: https://edc.intel.com/content/www/us/en/design/products/platforms/details/raptor-lake-s/13th-generation-core-processor-specification-update.