Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Local Approximation of NFV Workloads

Master Thesis of

Norbert Schmitt

At the Department of Computer Science
Chair for Computer Science II
Software Engineering

Reviewer:	Prof. Dr.-Ing. Samuel Kounev
Second reviewer:	Prof. Dr.-Ing. Phuoc Tran-Gia
Advisor:	M.Sc. Jóakim von Kistowski
Second advisor:	M.Sc. Piotr Rygielski

Duration: April 15th, 2016 – October 14th, 2016

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Würzburg, October 9, 2016

.....

(**Norbert Schmitt**)

Contents

Abstract - German

Mit dem zunehmenden Bedarf an Energie für die moderne IT Infrastruktur wird die Wahl effizienter Geräter immer wichtiger. Die SPEC stellt für diese Zwecke standardisierte Benchmarks bereit, die die Energieeffizienz messen. Zur Beurteilung der Energieeffizienz sind verlässliche Messungen realer Anwendungen notwendig. Durch die zunehmende Vernetzung von Systemen werden zuverlässige Messungen komplexer, da Software externe Eingaben benötigen kann um das Testsystem auszulasten. Dies kann unter anderem dazu führen, das Komponenten ausgelastet werden die unter anderen Workloads untätig sind und den Verbrauch reduzieren oder keinen Strom verbrauchen.

Zusammen mit dem steigenden Trend in Richtung Software Defined Networking (SDN) und Network Functions Virtualization (NFV), der Virtualisierung dedizierter Geräte als Softwarelösungen auf handelsüblichen Servern, erschwert die zuverlässige Messung extern unter Last gesetzter Software zusätzlich und erhöht die Fehleranfälligkeit.

In dieser Arbeit wird deshalb das Performance Event Trigger Framework (PET) vorgestellt. Um das Testsystem unter verschiedenen Laststufen zu betreiben und den Stromverbrauch zu approximieren löst es Performance Events aus. Das Framework ist als Chauffeur Bechmark konzipiert um eine einfache Nutzung zu gewährleisten. Zuerst werden die Performance Counter anhand der Korrelation mit dem Stromverbrauch ausgewählt und die Implementierung der Auslöser vorgestellt. Jeder Auslöser wird darauf auf seine Genauigkeit hin untersucht und in das Framework eingepflegt. Das Framework wird danach durch drei Workloads, dem Pi Workload aus Chauffeur, XMLValidate und SSJ aus SERT, evaluiert um die Machbarkeit und Genauigkeit der Approximation zu zeigen. Um zu zeigen das PET auch extern unter Last gesetzte Workloads annähern kann, wird ein vierter NFV Workload in Form einer DPI Firewall verwendet. Zusätzlich wird ein lineares Regressionsmodell erstellt zur weiteren Auswertung.

Es konnte im Rahmen der Arbeit gezeigt werden, dass die Näherung von Workloads anhand von Performance Countern möglich ist und das PET eine durschnittliche Genauigkeit von unter 10 % erreicht.

Abstract - English

With the growing demand of energy in modern IT infrastructure, selecting more efficient devices becomes more important as well. Standard benchmarks are provided by the SPEC to measure the energy efficiency. Assessing the energy efficiency requires reliable measurements of real world applications. Due to the rising connectivity between systems, reliable measuring energy efficiency becomes difficult. Workloads relying on external input might stress hardware components of a system which would otherwise lie dormant and reduce their consumption or not using any power at all.

Together with the rising trend of Software Defined Networking (SDN) and Network Functions Virtualization (NFV), virtualizing dedicated appliances as software running on off-the-shelf servers, reliable measuring efficiency on externally driven workloads becomes even more challenging and error prone.

This thesis therefore proposes the Performance Event Trigger Framework (PET). It triggers performance events to approximate a workload, which normally needs external devices to run the System Under Test (SUT) under different load levels. The framework is implemented as a Chauffeur benchmark to allow for easy use. First the performance counters that have a high correlation with power consumption are selected and an implementation to trigger these events is presented. Each event trigger is then evaluated in terms of accuracy and incorporated in the final framework. The framework itself is evaluated against three workloads, the Pi workload from Chauffeur, XMLValidate and SSJ from SERT to show the feasibility and accuracy of the approximation. A fourth NFV workload in form of a DPI firewall is also evaluated to show that PET can approximate externally driven loads. A linear regression model is also used as a second mean of evaluation if PET is able to approximate workloads.

In the course of this thesis it has be shown that approximating workloads with performance counters works and that PET has a reasonable accuracy, with average deviation of less than 10 %.

1. Introduction

Efficiency is a growing concern in today's IT infrastructure. In 2006 data centers in the United States consumed an estimate of 61 billion kWh annually. The energy consumption has risen to an estimated 93 billion kWh in 2013 and is projected to climb to 140 billion kWh by 2020, according to the National Resources Defense Council (NRDC) [WD14]. With the rising amount of servers and therefore energy consumption, the demand for network equipment will climb as well.

Reducing the power consumption and improving energy efficiency requires reliable measurements of real world applications the servers are executing. With increasing connectivity and dependencies between different applications, reliably measuring energy efficiency becomes difficult as benchmarks lack the ability to stress hardware components only used by interconnected workloads. The workloads must be driven by external load generators.

Virtualization, a growing paradigm in IT, transferred into the networking infrastructure with the introduction of Software Defined Networking (SDN) and later Network Functions Virtualization (NFV) in 2013. Therefore the energy efficiency of virtualized networks has become a considerable issue as well.

Often performance counters are used to estimate the power consumption of a system. Also the modeling via performance counters is widely researched. Yet the current use case scenarios for performance counter lack the ability for experimental measurements in case a workload is dependent on external appliances.

A lot of effort has been put into making data centers more energy efficient. One aspect is the measurement of server energy efficiency with the Server Efficiency Rating Tool (SERT), developed by Standard Performance Evaluation Corporation (SPEC) [LTA+12]. Rating the energy efficiency through benchmarking helps to provide an aid in selecting the most efficient server for the intended service or workload. Yet the SERT can only run locally and not stress hardware that would otherwise be used by workloads driven through external requests, such as a Virtual Network Function (VNF). This unused hardware could become active in real world scenarios and influence hardware components already evaluated for energy efficiency, which leads to a change in power consumption and hence reduced efficiency. Rating the energy efficiency of externally driven workloads also demands a more complex testbed as the load generators must be considered in the benchmark design. As a consequence, measurements and setup are more time consuming and error prone.

As modern servers are working under highly variable load intensities [vKHK14] and a server is not generally more efficient under full load [vKBB+15], benchmarking the server

under different load levels is necessary for a complete energy rating. Auto calibrating a load level aware benchmark that needs to be driven externally is challenging and not supported by the SERT. Additionally, external power meters used for power measurements need stable loads for higher accuracy [LT11]. Therefore the normally used workloads are often not suitable.

A lot of work has been done in terms of energy efficiency benchmarking and modeling. Often established benchmark suites like SPEC are used to achieve comparable results. Yet externally driven workloads are not part of these suites. In the NFV context, benchmarking is mostly focused on a broader scope, taking into account networks consisting of several entities while this thesis aims to abolish the need for complex interacting systems.

Models based on performance counters work, but mostly use local workloads for validation. If the models can predict highly interconnected applications accurately needs to be determined. Other works on performance counters only focus on the counters themselves and their accuracy and overhead for making them available to the user and not on how an accurate counter value can be achieved by deliberately triggering them.

To be able to measure energy efficiency of a complex workload that is request driven, the implementation and evaluation of a local approximation is proposed. It has been shown through models that performance counters and power consumption can be correlated. As a result, performance counter events, generated through software implementations, will be used to modify the counters accurately and achieve a power consumption approximation. The proposed approach remedies the need for external load generators, making efficiency measurements on externally driven workloads possible and simplifies the test setup. It can also be used to validate power or efficiency models on more complex workloads. The resulting software should be modular to make the framework extensible for future implementations of performance event triggers.

Before the proposed framework can be implemented, the correlation between the available performance counters and power consumption are calculated and optimal counters are selected. Multiple implementations and configurations for each counter are evaluated to determine the accuracy of triggering events and side effects on other counters. After the identification of the optimal implementations, the framework is created and evaluated against reference workloads. The reference workloads consist of a simple benchmark and workloads from the SERT as well as an externally driven workload in form of a Deep Packet Inspection (DPI) firewall (VNF). During all measurements, power consumption and the performance counters are measured and compared to the reference workloads. A regression model is used as a second validation method to test if the framework can accurately approximate externally driven workloads.

2. Foundations

This chapter introduces the required foundations for this thesis. First NFV is introduced to give an overview of its components. Afterwards a short description of the SERT measurement methodology is given on how measurements in this thesis are taken. This section also contains a description of the Chauffeur power and performance benchmarking application used as test harness. An introduction to performance counters and the IntelPCM tool used for instrumentation is provided. The chapter finishes with an explanation of preexisting workloads used for evaluation.

2.1. Network Function Virtualization

NFV describes the virtualization of network functions normally carried out by dedicated network appliances. Using dedicated hardware has certain drawbacks as stated by [CCW+12]. These include increasing costs of energy, decrease in hardware life cycles due to innovation and rare skills needed to integrate and operate highly specialized equipment.

It was introduced to counter the negative aspects by performing the network functions not on dedicated hardware but in software. The software can then be deployed more easily on commodity hardware. Configuration of a network and its functions is faster and easier as the software components of a NFV, the VNFs, can be remotely deployed without having to switch out network appliances at location. This makes it suitable for a wide range of use cases in the data and control plane as explained in [CCW+12].

The European Telecommunication Standardisation Group (ETSI) provides several use cases that benefit from having network functions virtualized instead of dedicated devices as stated in [ETS13].

- Network Functions Virtualization Infrastructure as a Service
- Virtual Network Function as a Service (VNFaaS)
- Virtual Network Platform as a Service (VNPaaS)
- VNF Forwarding Graph
- Virtualization of Mobile Core Network and IP Multimedia Subsystem (IMS)
- Virtualization of Mobile base station
- Virtualization of the Home Entertainment

- Virtualization of Content Delivery Networks (CDNs) (vCDN)
- Fixed Access Network Functions Virtualization

According to [ETS14], NFV consists of the following components. The main parts and interaction with each other can be seen in Figure 2.1.

1. **NFVI**
   Network Functions Virtualization Infrastructure (NFVI) is providing computational and virtualization resources for NFV to execute VNFs. The host for VNFs can be any device capable of hosting a Virtual Machine (VM) but is usually a commodity server.
2. **VNF**
   This is the software network function that can be executed in a VM like Network Adress Translation (NAT), load balancing or firewalling.
3. **NFV Management and Orchestration**
   Administration tools and protocols for the NFVI as well as VNFs. It is able to deploy new VNFs and reconfigure, migrate and monitor running Virtual Network Function Instances (VNFIs).
4. **Operation Support System (OSS)/Business Support System (BSS)**
   The OSS or BSS is operating and managing the supporting systems of the NFV. Such as the physical network or SDN.

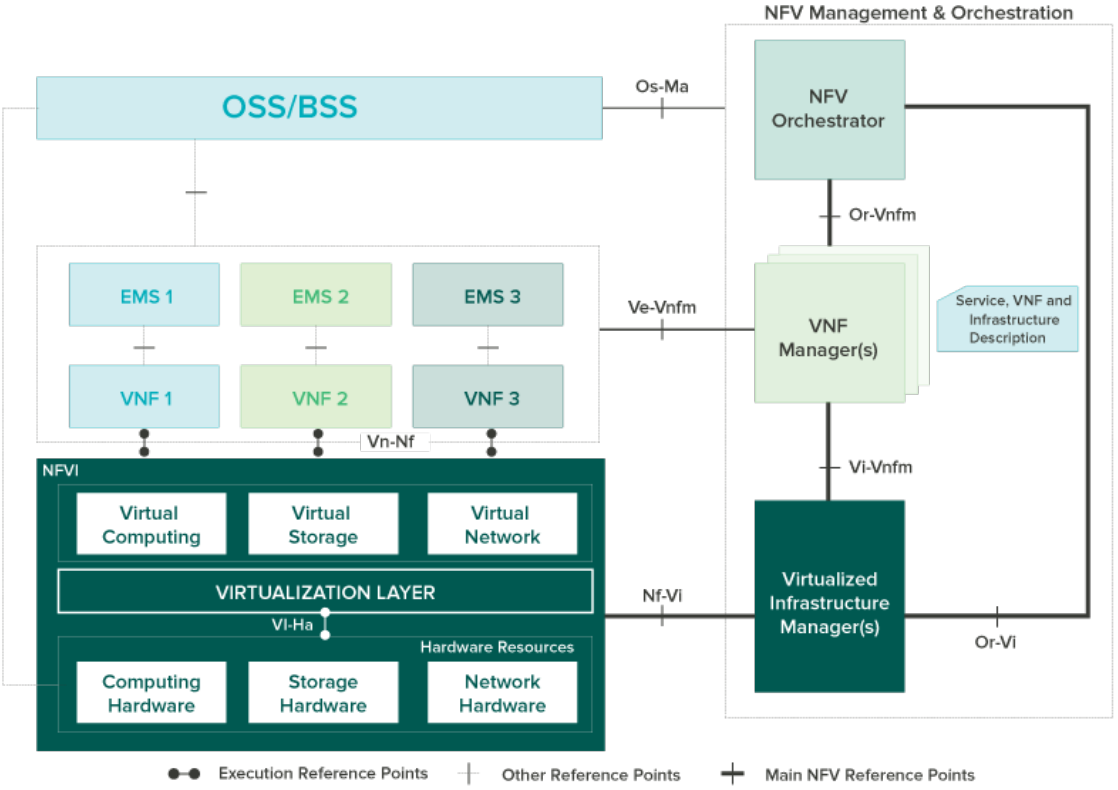


Figure 2.1.: *NFV architecture as taken from [OPN]*

The main component for this thesis is the VNF, in the form of a simple DPI firewall. This specific exemplary workload is selected as it stresses the System Under Test (SUT)'s Network Interface Cards (NICs). A NIC is a component usually not targeted by locally

running workloads. To determine the influence of externally activated hardware on the proposed approximation, the DPI firewall is selected as request driven reference workload.

An instance of a VNF deployed by a VNF Manager is referred to as VNFI. It can either be executed in a virtualized environment or directly on hardware. If executed virtualized, then each VNFI runs in its own virtualized container. VNFs are the basic building blocks of a NFV. A VNF can be build out of several Virtual Network Function Components (VNFCs) as shown in Figure 2.2 or as a single entity. If a VNF is build from multiple VNFCs, their internal interfaces do not need to be exposed. The VNF on the other hand has a well-defined interface to other network functions which can be either VNFs themselves or dedicated appliances. The implemented outside interface is defined by standardization organizations like the 3rd Generation Partnership Project (3GPP) or the Internet Engineering Task Force (IETF).

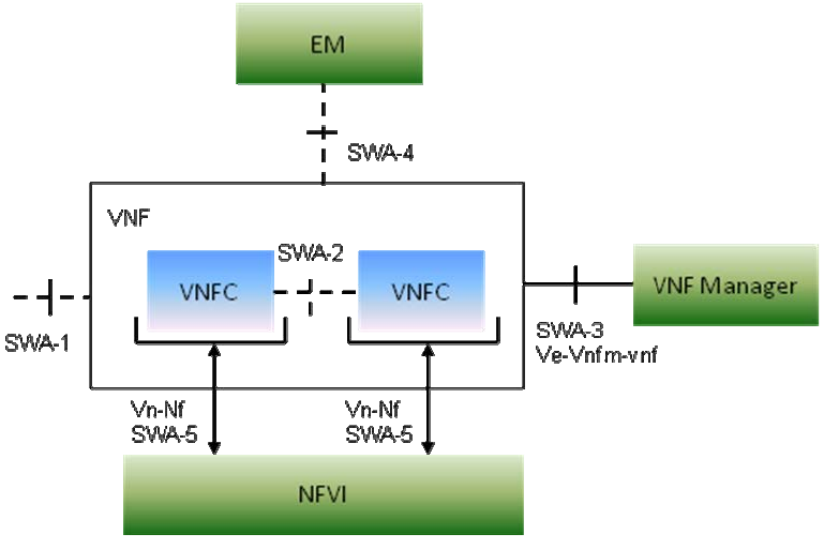


Figure 2.2.: *VNF functional view, taken from [ETS14]*

The requirements for a VNF deployment are described in the Virtual Network Function Descriptor (VNFD) of which each VNF has exactly one. The VNFD also includes descriptions of performance, security, reliability and other non-functional assurances as stated in [ETS14].

2.2. Power Measurement and Methodology

This section describes how measurements are taken. It is based on Power and Performance Benchmark Methodology [SPEb] and the Server Efficiency Rating Tool Design Document [SPE13] by SPEC. While SERT in its entirety itself is not used in this thesis, its guidelines on how energy efficiency is measured still apply to keep the measurements comparable as SERT is used as regulatory standard by the United States Environmental Protection Agency (EPA). It is also adopted by the industry like the SAP benchmark and Transaction Processing Performance Council (TPC) [SAP11] [PNV+10]. Most measurements are performed with the help of Chauffeur, which allows to automatically calibrate and run the workload at different load levels.

2.2.1. Measurement

All measurements in this thesis follow the methodology introduced by SPEC for benchmarks capable of graduated throughput levels. Throughput is defined as the number of

transactions executed in a fixed time frame. A transaction is one work package processed by the benchmarked workload. In case of the DPI firewall, one transaction equals one network package inspected. Transactions must have blocking behavior to get feedback during measurements of the number of transactions executed.

Benchmarks using graduated throughput levels request a steady state of transactions performed during a fixed time period. The steady state is enforced through the benchmark harness or the benchmark itself. The throughput levels are also referred to as load levels.

The sequence for such a benchmark is as follows, cited from [SPEb]:

1. System is made ready for measurement.
2. Harness starts environmental measurements
3. If required, initiate calibration process to determine maximum throughput
4. Compute intermediate measurement targets relative to maximum throughput
5. Iterate:
    a) Harness starts benchmark segment run at throughput interval X, where X begins at the highest target throughput and reduces each iteration until a zero-throughput interval can be measured, to obtain an Active-Idle measurement
    b) Delay as needed for benchmark synchronization and to achieve steady state
    c) Harness starts power measurements
    d) Harness or benchmark collects power and performance metrics.
    e) Harness ends collection of performance and power measurements
    f) Delay as needed for benchmark synchronization
    g) Benchmark segment completes
    h) Harness delays as needed for synchronization
6. Harness ends environmental measurements
7. Harness post-processes performance and power data

The basic SERT components are illustrated in Figure 2.3. The SERT test suite consists of different benchmarks, called worklets. The worklets dispatches transactions to put the SUT under load. These worklets are grouped together in workloads. Workloads are designed to stress certain aspects or components of the SUT. The contained worklets are run in a defined sequential order. All workloads together comprise the test suite and are also run sequentially. A worklet can also have a sequence of multiple different transactions. The transactions in a worklet sequence are executed one after another in a fixed order.

Worklets are further divided into three sequences as shown in Figure 2.4. First the *warmup sequence*, secondly the *no delay sequence* or calibration and finally the *graduated measurement sequence* which stresses the SUT under the defined load levels. The warmup and calibration sequences are optional to a worklet.

Each sequence is divided into intervals. An interval is comprised of three phases. The first phase is the *pre-measurement*, which lets the SUT settle for the current load level and reduce influences of oscillating startup behaviors of certain hardware or software components. In the next phase, *measurement* or *recording*, power measurements are taken and the actual throughput is measured. An interval finishes with the *post-measurement* phase. Each phase runs for a fixed time frame.

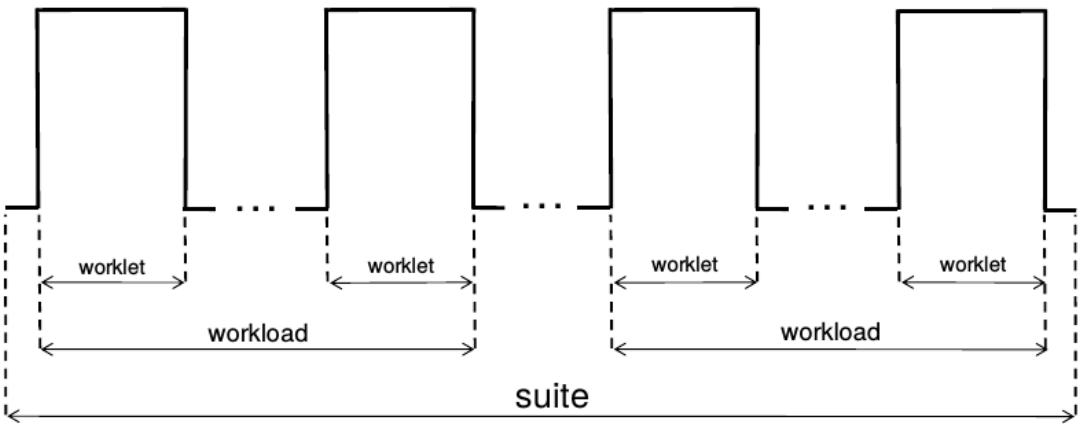


Figure 2.3.: *SERT components, as taken from [SPE13]*

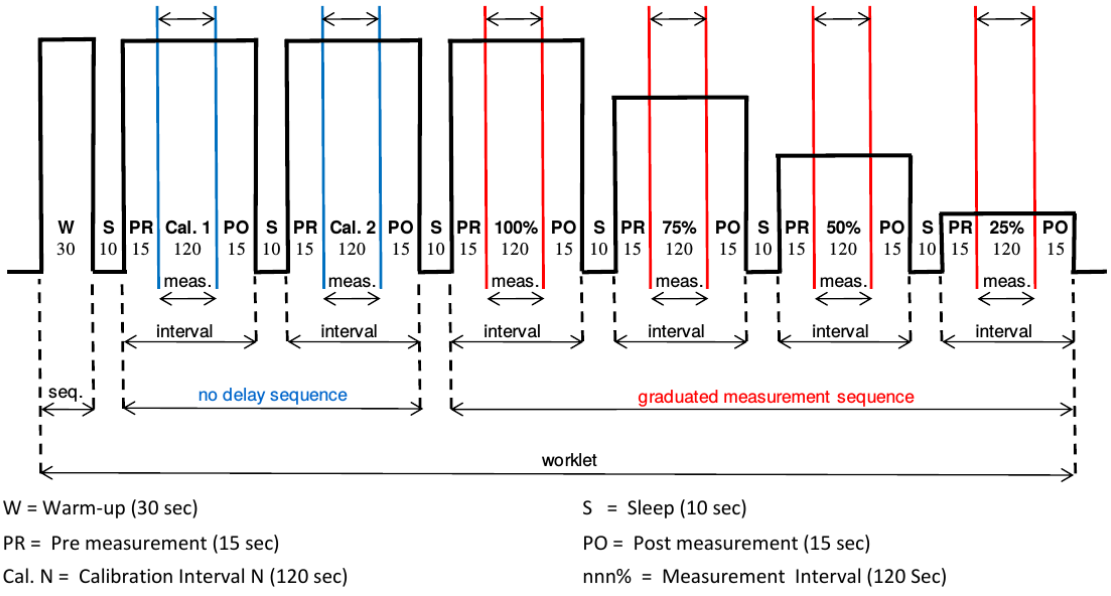


Figure 2.4.: *Worklet measurement execution, taken from [SPE13]*

SPEC allows several methods for calibrations, as long as the benchmark can be run consistently over the measurements. Four options are pointed out specifically by [SPEb]. The first three are of special interest because they allow for automatic calibration. The fourth option is for engineering and academic environments.

- The benchmark is run once at maximum throughput. The actual resulting throughput is used as maximum.
- The benchmark is run 2 to n times and the last two runs are averaged. The resulting value is the maximum throughput.
- The benchmark is run multiple times until the last runs result is lower then the preceding one. If this is the case, run the benchmark a last time and average over the last three runs to compute the maximum.
- Set the maximum throughput to an arbitrary fraction of the benchmark run.

As part of this thesis is to simplify testbeds for externally driven loads, including autocalibration, the fourth option is not selected. Because the SERT uses the second option and to evaluate the proposed framework as closely to existing benchmark runs as possible, the second option is selected.

2.2.2. Chauffeur

Chauffeur is used as a test harness and this section is mainly based upon the Chauffeur^{TM} Worklet Development Kit (WDK) User Guide [SPEa] and [LAB+13]. Chauffeur incorporates the aforementioned measurement methodology and is also used by the SERT. It is designed with the following principles in mind.

- Energy Measurements
- Scalability
- Ease of use
- Portability
- Flexibility

Figure 2.5 shows an overview of Chauffeur. It consist of the following hardware (HW) and software (SW) parts:

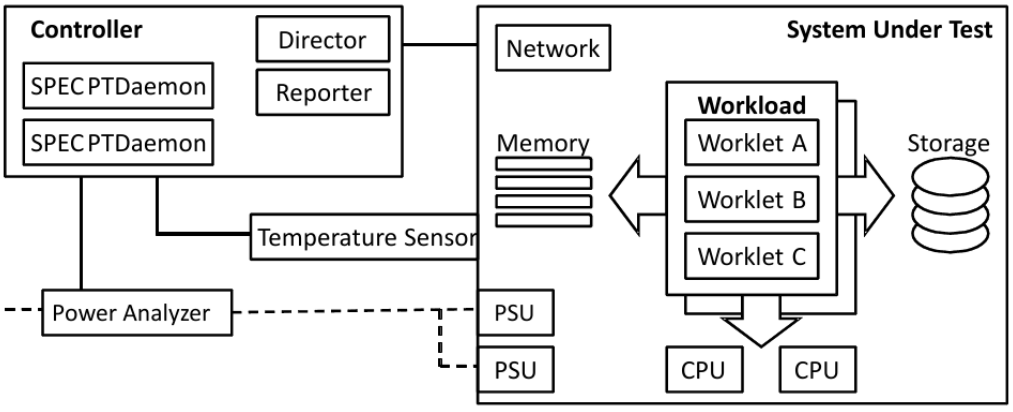


Figure 2.5.: *Chauffeur overview from [SPEa]*

Controller (HW)	The controller system governs which sequence is run and for how long. It also collects the measurement data. It communicates with the SUT and Power Analyzer via TCP/IP.
SUT (HW)	The actual system stressed during measurements.
Power Analyzer (HW)	Measures the power consumption of the SUT.
Temperature Sensor (HW)	Measures the ambient temperature in the SUT's vicinity.
SPEC PTDaemon (SW)	Collects the power and temperature measurements from the Power Analyzer and Temperature Sensor.
Director (SW)	Controls the measurement sequence on the SUT.
Reporter (SW)	Receives all results from a measurement and compiles a report of the benchmark suite.
Workload (SW)	Actual workload stressing the SUT. It is controlled by the Director.

The Chauffeur components and communication are illustrated in Figure 2.6. The Director runs as a Java Virtual Machine (JVM) on the controller and communicates with the SPEC PTDaemon to collect power measurement results. It also controls the Host JVM which in turn runs the Clients. Chauffeur can be configured to run the workload on several Central Processing Unit (CPU) cores in parallel. For each core it should run on, a Client JVM is started by the Host which actually runs the workload and worklets, and dispatches transactions until the current interval is finished. The results are collected and send to the Director, which is handing them on to the Reporter compiling the final reports. All communication is done via TCP/IP.

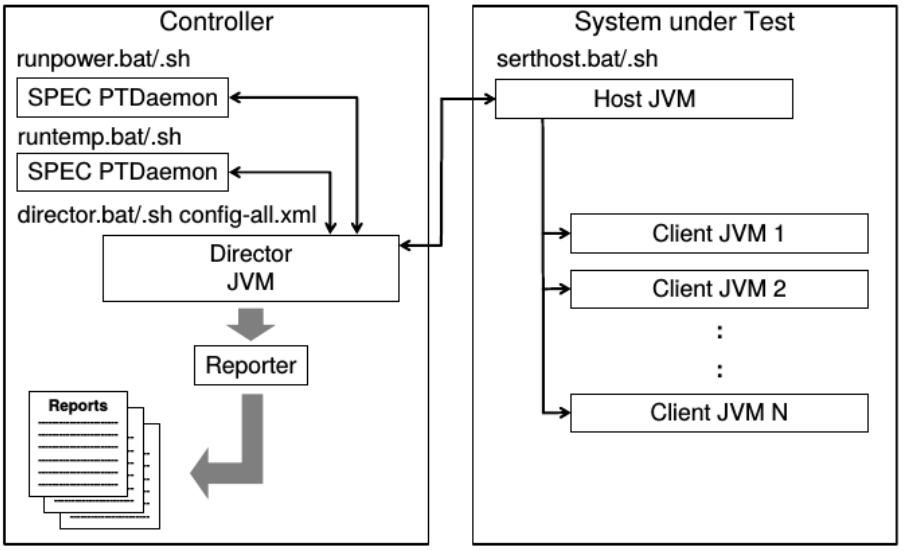


Figure 2.6.: *Chauffeur components and communication from [SPEa]*

2.3. Performance Counters

This thesis tries to model a workload's power consumption locally by generating performance counter events. Performance counters are integrated in today's hard- and software and can be read with either specialized software like IntelPCM, performance monitoring utilities of the Operating system (OS) or manually by directly accessing CPU registers.

This chapter will first give an introduction of what performance counters are and some drawbacks associated with them. Afterwards the counters available on the SUT are listed in their respective sections of either CPU counters which are read with IntelPCM or operating system counters that will be read by using the Linux virtual filesystem `proc`. This work focuses on performance counters available on Intel platforms due to the available testbed which uses an Intel Xeon E3-1230 v5.

The following descriptions are based on [Int16b] and [AMD16]. To monitor a systems performance and behavior, hardware manufacturers including Performance Monitoring Units (PMUs) for observation. A PMU includes Model Specific Registers (MSRs) which can store the count for either of the two event types:

- *Occurrence event* counts the number how often an event has been observed.
- *Duration event* counts the accumulated clocks for which an event has been observed.

Performance counters are further divided into two different classes, *architectural* and *non-architectural*. Architectural counters are a smaller subset of events as they are available across processor implementations. Non-architectural counters offer a wider variety of countable events but are model specific and therefore might not be available on certain systems.

The PMU also includes a Pin Control (PC) flag to toggle Performance Monitoring (PM) pins (PM0/BP0 or PM1/BP1) if an event occurs or an MSR overflows. PM events cover a broad scope on what the hardware is currently doing, which includes for example, but is not limited to, cache hits or misses on each cache level, falsely or correct predicted branches and instructions retired. The recorded performance counter events can be read on a system wide basis. Subsets can also be read on a per socket or a per core basis.

Monitoring performance events using a PMU does have its disadvantages. Weaver et al. [WTM13] have shown that modern implementations of performance counters are not accurate and have a tendency to deviate. They found two causes of deviation that make interpreting performance counters a difficult task. One cause is *nondeterminism* in which identical workloads resulted in different counter values and *overcount* for which counters where increased multiple times for the same instruction. Another problem presents itself if the PM pins are used. For occurrence events, it might be possible that an event happens twice within one clock cycle. While the value in the MSR is incremented twice, the pin is only asserted once which causes a deviation.

IntelPCM is a tool provided by Intel for monitoring performance counters. It can either be executed as a standalone application or integrated into a program using its C++ Application Programming Interface (API). As reading performance counters requires careful implementation, IntelPCM is selected as tooling to minimize errors resulting from erroneous self implemented instrumentation.

The operating system itself also counts certain events that can be obtained. In the case of Linux, these counters can be read from the virtual `proc` file system. Especially the files `/proc/stat` and `/proc/meminfo` are of interest. They, for example, give insight on how many hard and software interrupts have been processed, the number of context switches performed and memory utilization.

2.4. Preexisting Worklets

For evaluation, three readily available worklets are selected as a basis to test if the framework works for approximating local applications. They were selected as they stress different hardware parts of the SUT. The first worklet described in this section is the π worklet,

approximating π. It was selected as it gives a first evaluation if the proposed framework can approximate simple workloads and using performance counters is a viable approach. The Pi worklet is followed by XMLValidate from the SERT which stresses the CPU and memory, making it more complicated, provoking more counters to be triggered to validate the framework with a wider variety of performance counters. The last worklet described is the hybrid SSJ worklet, also from the SERT. It is closest to a real world application and combines different worklets stressing a more diverse set of hardware components. The evaluation of this workload will show if the framework can approximate locally running workloads reasonably well before evaluating externally driven workloads.

2.4.1. Pi Worklet

The Pi worklet comes from the ChauffeurTest package shipped together with Chauffeur. It calculates π with an iterative approximation using the Gregory-Leibniz series shown in 2.1. The upper limit for n is set randomly for each transaction with values between $[1000; 100000]$. This worklet is especially heavy on the CPU but not other hardware components of the SUT. It is therefore a good basis worklet for a first evaluation if the proposed framework is feasible.

$$\pi = 4 \cdot \sum_{k=1}^{n} \frac{(-1)^{k+1}}{2k-1} \tag{2.1}$$

2.4.2. XMLValidate

The XMLValidate worklet, as described in [SPE13], uses the Java `javax.xml.validation` package to implement transactions. The worklet validates Extensible Markup Language (XML) files against a XML schemata as shown in Figure 2.7. It randomizes the input file by swapping commented regions within the XML file. Despite used as a CPU benchmark, it also stresses the memory subsystem. It also performs a more complex operation than the Pi worklet. XMLValidate is used as feasibility evaluation when the memory system is stressed as well.

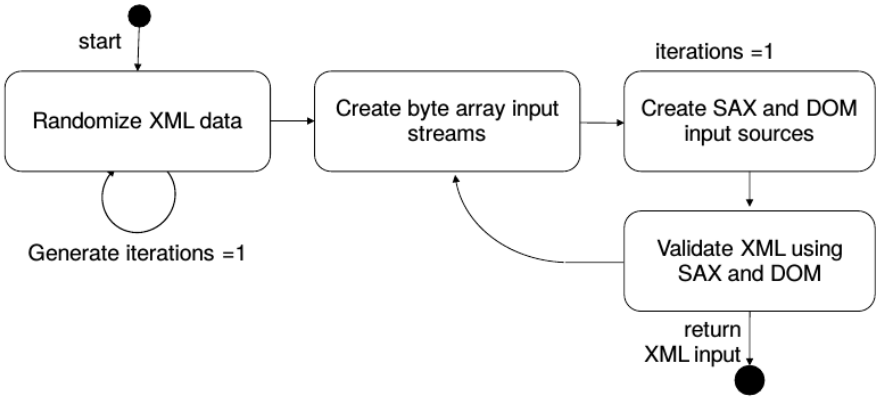


Figure 2.7.: *XMLValidate transaction from [SPE13]*

2.4.3. SSJ

SSJ from the SERT test suite simulates Online Transaction Processing (OLTP) as a Server Side Java application. It is described in [SPE13] and based on SSJ in SPECpower_ssj2008 but is not comparable to it. SSJ is a hybrid worklet, which means it is stressing multiple

components of the SUT. The CPU, caches and memory of the SUT are stressed. This worklet is closer to a real world application than the Pi or XMLValidate worklets. It is therefore raising the complexity in the evaluation of the proposed approach.

The SSJ worklet includes six transactions as a worklet sequence which are executed with certain frequencies as cited from [SPE13]. The mentioned frequencies are approximations how often a transaction occurs.

New Order (30.3 %)	A new order is inserted into the system
Payment (30.3 %)	Records a customer payment
Order Status (3.0 %)	Requests the status of an existing order
Delivery (3.0 %)	Processes orders for delivery
Stock Level (3.0 %)	Finds recently ordered items with low stock levels
Customer Report (30.3 %)	Creates a report of recent activity for a customer

3. Related Work

A number of work has been in done in terms of energy efficiency, power consumption, NFV performance counters. These topics often overlap. The following chapter will give an overview on the specified topics. First the aspect of efficiency measurements and improvements, followed by a short overview of work on NFVs. Finally a summary of performance counter and their usage in power estimation and modeling is presented.

3.1. Energy Efficiency and Benchmarking

A wide variety of work has been done in terms of energy efficiency benchmarking with a focus on CPU loads and NFVs. Most work focuses on a broader scope like [vKBL+15], in which workloads are distributed hierarchically from multiple machines down to Simultaneous Multithreading (SMT) and also Bagaa et al. [BBLM14], which distributes VNFs efficiently while also taking SDN into account. Botero et al. [BHD+12] developed a Mixed Integer Program (MIP) for consolidating VNFs in a virtualized network to achieve better energy efficiency. A. Beloglazov and R. Buyya [BB10] improve efficiency by migrating VMs within a data center while ensuring Quality of Service (QoS). However, in the aforementioned works, the efficiency is only viewed in the scope of a complete network, taking into account several systems or VNFs instead of a single entity.

Work from Jin et al. [JWC12] concentrates on the trade-off between the virtualization overhead and efficiency increase using VMs by aggregating workload on fewer physical machines with higher utilization. Yet using less systems does not necessarily increase efficiency, as shown in [vKBB+15], in which the efficiency of the SERT CPU workloads under different load levels is measured. It also explores the influence of different workloads on efficiency. The proposed solution on the other hand tries to simulate workloads on a lower level without the need to approximate multiple connected systems.

A more focused paper [vKBB+16] shows that repeatable power measurements can be difficult to achieve and measurement results can vary, even with nominally identical CPUs. Yet this thesis is not only measuring CPU efficiency but rather a complete system, in which the CPU is only one of many components influencing the efficiency.

It is evident that this thesis is in between both categories of having a broad view of connected systems and focusing on specific components.

3.2. NFV Deployment

Work on NFVs focuses often on scaling a NFV to provision for dynamically changing demands. The work of Moens and DeTurk [MDT14] as well as Mijumbi et. al. [MSG+15] are concerned with deploying VNFs in virtualized environments. While both develop models and algorithms for efficient VNF placement, they do not take into account power consumption. In an article from Bouet et. al. [BLCC15] a cost minimization heuristic is developed which takes different aspects of VNF placements into account. Yet the energy efficiency or power consumption, a central part of this thesis, is not addressed.

3.3. Power Estimation and Performance Counters

Performance counters are used for performance analysis of software. One way for such an analysis is calculating Call per Instructions (CPIs) stacks. In a paper from S. Eyerman et al. [EEKS06] performance counters are used to calculate more accurate stacks on superscalar out-of-order processors. They use different miss events like cache misses and Translation Lookaside Buffer (TLB) misses for their CPI stack calculation.

Performance counters are not only used to analyze performance, but are also useful for compiler optimizations through an off-line learning model, as shown in [CFA+07]. In the paper from Singh et al. [SBM09], a model derived from performance counters is used to implement a thread scheduler that takes power consumption into consideration. Another paper from F. Bellosa [Bel00] also describes thread scheduling by enriching the thread context with performance counter and known energy values. Having a framework that can reliably trigger performance events can be used for validation and testing such implementations.

Modeling the power consumption based on performance counters is also a possible application. In the papers from Bircher and John [BJ12], Lewis et al. [LGT08] and Isci and Martonosi [IM03], models are developed estimating the power consumption as a function of performance counters dependent on the workload. The work from Contreras and Martonosi [CM05] also build a model based on performance counters but focuses on embedded devices with a specific CPU and memory. Kadayif et al. [KTK+01] provide a tool based on performance counters for the UltraSPARC platform which provides energy estimations. They all show that performance counters can be used for power estimations. It is therefore expected that the proposed framework for approximating power consumption by triggering counter events is a viable approach and can help in model validation and test cases for tooling and instrumentation.

In a paper from Zaparanakus et al. [ZJH09], an overview on the accuracy of the measurement infrastructure for multiple CPUs is presented. Weaver et al. [WTM13] for instance identified two major deviations, which can influence counter values. Weaver also researched the overhead of common performance counter implementations and found that the current PAPI interface has large overheads [Wea15]. Despite research in the area of performance counter accuracy, none of the mentioned works treated the accurate generation of counter events.

4. Approach

This thesis aims to develop a framework leveraging performance counters that can approximate workloads locally which would usually be driven by external load generators. The approach described in this section shows how the proposed approximation is measured and evaluated.

Before an implementation can be made, first the most relevant performance counters have to be identified. A correlation analysis is made on the reference measurement of the NFV workload. It is the most diverse workload and also stresses the SUT's NICs.

To prove that an approximation is possible, multiple implementations are made with the purpose of triggering events. Each event trigger is then evaluated if it can accurately generate performance counter events, reaching the target value without over or under counting. As counters might be able to introduce side effects on other counters that are implemented, it is necessary to identify them and incorporate these side effects in the framework.

After the most viable implementation for each event trigger is selected, the framework is build and evaluated against the local benchmarks described in Section 2.4. The power consumption of the local workloads is measured under different load levels and the performance counters are recorded. If the framework is able to approximate the local references, it shows that simulating power consumption through performance counters is a reasonable approach.

During all reference and evaluation measurements, performance counters are recorded for evaluation against a linear regression model, based on the preexisting local reference measurements. It is used as a second evaluation step for the framework's approximation performance.

4.1. Reference Workloads

This section describes the purpose of the selected workloads. The local workloads introduced in Section 2.4 are used for feasibility while the NFV workload acts as the externally driven application.

4.1.1. Local Workloads

To act as reference if workloads can be approximated through performance counters, three locally executable workloads stressing specific parts of a SUT with increasing complexity are selected.

- *Pi worklet*: Simple approximation of π. Mainly stresses the CPU.
- *XMLValidate*: Validates XML file according to a schemata. Stresses both the CPU and memory.
- *SSJ*: A simulation of OLTP, resembling a real world application. This worklet put CPU, memory and caches under load.

These workloads are not dependent to be driven by an external load generator. They therefore do not have I/O heavy operations resulting from network traffic or otherwise. Yet they stress certain aspects of SUT to evaluate the feasibility of the proposed approximation framework. Measurements are performed as described in Section 2.2.1.

4.1.2. NFV Workload

The NFV workload, described in more detail in Section 5.2, acts as the externally driven reference workload. As VNFs are stressing the network adapter, which is not used by the local workloads, a VNF is a good example for externally driven workloads. The selected VNF is a DPI firewall inspecting User Datagram Protocol (UDP) packets and checking them for validity.

To put the SUT under stress, network traffic needs to be generated. A traffic generator is implemented that produces suitable traffic and is able to saturate the available network connection. The generator must also be able to run at different load levels like the Chauffeur harness for measurements to be comparable. A receiver is installed, to which allowed packets are routed to ensure the VNF is working correctly and can be calibrated.

One of the main problems in calibrating the testbed is the non-blocking behavior of the implemented transactions (packets inspected by the DPI). This is inherent to most externally driven workloads as no knowledge about the SUT's current transaction status is available. To perform a reference measurement with actual external traffic, calibrating the VNF deviates from the procedure introduced in Section 2.2.1. The maximum throughput is defined as the number of packets handled by the SUT without dropping packets due to overload. A disadvantage is that the overload is likely to happen because the network bandwidth is saturated instead of the SUT's CPU and memory. Calibration is done via binary search until the threshold of packets dropped is within reasonable bounds or the maximum number of steps is reached. Afterwards measurements are taken in the described manner, recording the following data:

- Valid packets send
- Malicious packets send
- Packets received
- Performance counters
- Power consumption

4.2. Workload Approximation

To achieve a workload approximation through performance counters, suitable candidates have to be identified from the available sources, IntelPCM and the OS. The available performance counters are presented and suitable counters are selected. Afterwards the possible influences of side effects when triggering events is described.

4.2.1. Selection of Available Performance Counters

A wide variety of performance counters are available on modern systems. Not only are counters measured by the CPU itself but also by the OS. To identify and select counters that have an influence on power consumption, the Pearson correlation between counters and the power consumption is calculated for different load levels also used in the evaluation. Ten load levels ranging from 10 % to 100 % in 10 % steps are used. The measurement from which the correlation is calculated is the VNF workload, as the proposed framework should be able to approximate externally driven workloads. A correlation below 0.8 is considered to low for implementation. Feasibility of implementation is considered as well as a second criteria. This is necessary because it might be possible that some performance counters might not be susceptible to manipulation from user space programs and need modifications of the OS to work, which is not within the scope of this thesis. On the other hand, some performance event triggers might be implemented with only minor changes to already selected counters, justifying a lower correlation in certain cases.

Performance counters with values that had a constant value of zero or listing hardware configurations during the measurement are omitted.

CPU Performance Counters

The correlations are presented in Table 4.1 and descriptions are cited from [Wil]. The C-State (CPU power saving states) residency counters are also omitted. This is due to external tools needed or a CPU specific self-written implementation to send the CPU to a defined C-State. This might not be executable on other hardware other than the SUT's. The QuickPath Interconnect (QPI) and L3 cache occupancy metrics are omitted because they are not measured in the IntelPCM configuration used for this thesis. CPU internal power metrics are omitted because the power consumption is measured externally. Temperature readings are not a part of this thesis.

The *EXEC* counter is not selected, despite high correlation, as it is dependent on *FREQ*. *IPC*'s correlation is not deemed sufficient for implementation. *FREQ* has high correlation but the framework should approximate a workload under real conditions in which the frequency is usually not influenced by user space programs. It is also dependent on *ACYC*. The same is true for *AFREQ*. Its high correlation is caused by a nearly constant value as a fraction of the CPU's design frequency. *L3MPI* and *L2MPI* are a byproduct of *INST* and *L3MISS / L2MISS* with low correlation. *ACYC* is well correlated, but not selected for the same reasons as *FREQ*. The *TIME(ticks)*, *PhysIPC* and *PhysIPC%* counters are not selected due to low correlation. *INSTnom* and *INSTnom%* are dependent on *INST* which is selected to be implemented.

Counters implemented that have a high correlation without directly modifying the CPU's configuration are *L3MISS*, *L2MISS*, *READ*, *WRITE* and *INST*. Despite lower correlation, *L3HIT*s and *L2HIT*s are selected. If a memory access misses L2, it could either hit or miss L3. It therefore seems reasonable to select *L3HIT* as L2 misses could directly generate L3 hits or misses if needed. As all other cache related performance counters are selected and the cache's content need to be controlled in a specific manner, *L2HIT* is therefore selected as well despite moderate correlation.

Linux Recorded Performance Counters

The performance counter correlations together with a description, taken from [BBN+] and shortened where necessary, are presented in Table 4.2 and 4.3.

For this thesis, *user* is not selected despite a high correlation value. The number of background programs cannot be influenced as the SUT is already running with a minimum

Perf. counter	Correlation	Selected	Description
EXEC	0.976		Instructions per nominal CPU cycle ignoring turbo and power saving modes
IPC	0.577		Instructions per cycle
FREQ	0.981		Frequency relative to nominal CPU frequency
AFREQ	0.992		Frequency relative to nominal CPU frequency excluding the time when the CPU is sleeping
L3MISS	0.968	✓	L3 cache line misses
L2MISS	0.983	✓	L2 cache line misses
L3HIT	−0.629	✓	L3 cache hits
L2HIT	0.727	✓	L2 cache hits
L3MPI	0.478		L3 cache misses per instruction
L2MPI	−0.693		L2 cache misses per instruction
READ	0.943	✓	Memory read traffic
WRITE	0.969	✓	Memory write traffic
INST	0.976	✓	Number of instructions retired
ACYC	0.981		Number of clockticks including turbo and power saving modes
TIME(ticks)	−0.203		Number of invariant clockticks invariant to turbo and power saving modes
PhysIPC	0.582		*IPC* multiplied by number of threads per core
PhysIPC%	0.576		*PhysIPC* relative to maximum *IPC*
INSTnom	0.976		Instructions per nominal cycle multiplied by number of threads per core
INSTnom%	0.976		*INSTnom* relative to maximum *IPC*

Table 4.1.: *CPU performance counters*

of software installed and the number of processes created by the test harness is fixed. The *softirq* counter is not selected even with high correlation because software interrupt are handled only after a system call or hardware interrupt [Pro06]. Therefore hardware interrupts are deemed sufficient. Yet it should not be discarded but implemented in the future to possibly improve the proposed framework. The *processes* counter has a good correlation but was not selected as the implementation for context switches (*ctxt*) creates threads which will interfere with this counter. As with the *softirq* counter, it should be kept in mind for further improvements. All other not selected counters correlation is deemed too low to be selected.

Hardware interrupts (*irq*) and the number of context switches (*ctxt*) are selected due to their high correlation.

Perf. counter	Correlation	Selected	Description
user	0.874		Normal processes executing in user mode
system	0.565		Processes executing in kernel mode
idle	−0.675		Processes currently idling
iowait	−0.322		Waiting for I/O to complete
ctxt	0.992	✓	Number of context switches
processes	−0.984		Number of processes and threads created
procs_running	0.685		Number of threads running or ready to run (runnable threads)
procs_blocked	−0.055		Number of processes blocked, waiting for I/O to complete
softirq	0.993		Software interrupts serviced since boot time
irq	0.997	✓	Number of interrupts serviced since boot time

Table 4.2.: *Linux performance counters*

The memory performance counters are also read. Yet none of them is selected for implementation. The reasons are low correlation and feasibility of implementation such as *SReclaimable*, which is is part of the in-kernel data structure cache *Slab*. The only two counters not falling into these two categories are *MemFree* with a correlation of 0.975 and *MemAvailable*, with −0.821. Test implementations are made to influence these counters but no noticeable impact could be measured. Further evaluation of these counters for future implementation is therefore necessary but is not within the scope of this thesis.

4.2.2. Side Effects of Triggering Performance Events

Triggering performance events to modify the counters can have side effects, especially counters for more abstract events such as context switches. But also counters closer to hardware do have side effects. A read operation on memory for instance could trigger cache misses or hits, depending on the cache lines currently residing in cache. Therefore side effects for each selected performance counters are evaluated by generating a large number of each events individually. The degree to which an event causes side effects on other counters is measured and included in the frameworks configuration.

To determine if side effects improve the approximation, two composition mechanisms are implemented for evaluation. First an accumulation method is implemented as shown in Equation 4.1. The accumulated side effects s_x affecting the event count v_x are calculated by multiplying the side effects for a single event s_i with the number of events causing the

Perf. counter	Correlation	Description
MemFree	0.975	The sum of *LowFree* and *HighFree*
MemAvailable	−0.821	An estimate of how much memory is available for starting new applications, without swapping. Calculated from *MemFree*, *SReclaimable*, the size of the file Least Recently Used (LRU) lists, and the low watermarks in each zone.
Buffers	−0.984	Relatively temporary storage for raw disk blocks
Cached	−0.984	In-memory cache for files read from the disk (the pagecache). Does not include *SwapCached*.
Active	−0.984	Memory that has been used more recently and usually not reclaimed unless absolutely necessary.
Inactive	−0.950	Memory which has been less recently used. It is more eligible to be reclaimed for other purposes
Active(anon)	−0.531	Not available
Active(file)	−0.984	Not available
Inactive(file)	−0.950	Not available
Dirty	−0.165	Memory which is waiting to get written back to the disk
Writeback	0.290	Memory which is actively being written back to the disk
AnonPages	−0.575	Non-file backed pages mapped into userspace page tables
Mapped	−0.721	Files which have been mmaped, such as libraries
Slab	0.627	In-kernel data structure cache
SReclaimable	−0.935	Part of *Slab*, that might be reclaimed, such as caches
SUnreclaim	0.714	Part of *Slab*, that cannot be reclaimed on memory pressure
KernelStack	0.587	Not available
PageTables	−0.043	Amount of memory dedicated to the lowest level of page tables
Committed_AS	−0.028	The amount of memory presently allocated on the system. The committed memory is a sum of all of the memory which has been allocated by processes, even if it has not been "used" by them as of yet. A process which `malloc()`'s 1 GiB of memory but only touches 300 MiB of it will show up as using 1 GiB.

Table 4.3.: *Linux memory performance counters*

effect v_i. The effects are summed over all event triggers i. If the resulting accumulated side effects are lower than v_x, they are subtracted and set as the new count of events to trigger including side effects $v_{s,x}$. In case s_x is higher than v_x, the new count is set to zero, triggering no events.

$$\begin{aligned} s_x &= \sum_{i=1}^{n} s_i \cdot v_i \\ v_{s,x} &= \begin{cases} v_x - s_x & \text{if } s_x \leq v_x \\ 0 & \text{if } s_x > v_x \end{cases} \end{aligned} \tag{4.1}$$

It is likely that the simple aforementioned method is not an optimal solution when balancing side effects and number of events to generate. An analytic solution to this problem might not be feasible to obtain. Therefore simulated annealing is used to find an optimal solution. Simulated annealing is selected as it is less prone to be caught in local maxima or minima like hill climbing. The following description is mainly based on [HJJ03] and adapted to the proposed framework.

In simulated annealing the global minimum solution ω^* is searched in the solution space Ω with $\omega^* \in \Omega$. To find the global minimum, an energy function $f : \Omega \to \mathbb{R}$ is needed to assess if a solution is closer to the minimum, while the condition $f(\omega) \geq f(\omega^*)$ must be fulfilled. As energy function, a modified Mean Squared Error (MSE) function as shown in Equation 4.2 is used. $\hat{\omega}_i$ is the target value for the i-th operation, ω_i the current value and $\omega_{s,i}$ the side effects imposed by the current configuration.

$$f(\omega) = \frac{1}{n} \sum_{i=1}^{n} (\hat{\omega}_i - \omega_i - \omega_{s,i})^2 \tag{4.2}$$

Each solution ω has neighbors defined by a neighborhood function $N(\omega)$. A neighbor is a solution that can be reached within a single iteration. A neighboring solution $\omega' \in N(\omega)$ is calculated by selecting a random operation ω_i and incrementing it by one. If a calculated neighbor is accepted based on the acceptance probability in Equation 4.3 with t_k as the temperature parameter or cooling schedule at iteration k. t_k is defined such that $t_k > 0$ for all k and $\lim_{k \to +\infty} t_k = 0$.

$$P\{\text{Accept } \omega' \text{ as next solution}\} = \begin{cases} e^{\frac{-\Delta_{\omega,\omega'}}{t_k}} & \text{if } \Delta_{\omega,\omega'} > 0 \\ 1 & \text{if } \Delta_{\omega,\omega'} \leq 0 \end{cases} \tag{4.3}$$

If the temperature T is reduced slowly, then simulated annealing can reach a steady state. The probability of the system in state ω with the energy $f(\omega)$ at temperature T follows a Boltzmann distribution. As distance metric $\Delta_{\omega,\omega'}$, shown in 4.4, is used.

$$\Delta_{\omega,\omega'} = f\left(\omega'\right) - f(\omega) \tag{4.4}$$

First the initial solution is selected, the temperature change counter is set $k = 0$. Afterwards the temperature parameter t_k and the initial temperature $T = t_0 \geq 0$ are selected. Finally before the algorithm can start, M_k is set which defines the number of iterations at each temperature. Then the algorithm is ready and can be started. It repeats the following steps until the stopping criterion is met by either finding the minimum energy or a predefined number of iteration steps were executed.

1. Set repetition counter $m = 0$
   a) Generate a solution $\omega' \in N(\omega)$
   b) Calculate $\Delta_{\omega,\omega'}$
   c) $\omega \leftarrow \omega'$ with probability from 4.3
   d) $m \leftarrow m + 1$
   e) If not $m = M_k$, then goto a)
2. $k \leftarrow k + 1$
3. If stopping criterion is not met, then go to 1.

4.3. Regression Model

As stated earlier, a linear regression model based on measurements from the local reference workloads Pi, XMLValidate and SSJ is created as as a second evaluation method. If the model is able to predict the power consumption generated by the framework, it shows that performance counters are able to approximate workloads independent of external appliances. If the prediction for the NFV workload is false, but correct for local workloads, it gives an indication that hardware used by externally driven workloads can significantly influence the approximation when trying to simulate them. In case the local and NFV workloads are incorrect, it shows that the used modeling approach is not sufficient for approximating power consumption through the selected performance counters if the measurements show a working approximation.

The model is generated using multiple linear regression. The following section is based on [FKLM13]. The power consumption y is modeled through the input parameters, the performance counters, $x_1, \ldots, x_k$. y is therefore modeled as the conditional expected value $E(y|x_1, \ldots, x_k)$ and can be written in a general form as shown in Equation 4.5.

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k + \epsilon \tag{4.5}$$

The model parameters $\beta_0, \beta_1, \ldots, \beta_k$ need to be estimated from the reference measurements. The first parameter β_0 is the intercept. Both x and β can be written in vector form, including the intercept. So we obtain $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_k)$ and $\boldsymbol{x} = (1, x_1, \ldots, x_k)'$. The Equation 4.5 can now be written as $y = \boldsymbol{x'\beta} + \epsilon$.

Performance counter	Estimated coefficient β	Standard Error
Intercept	23.214	1.1483
L3 misses	$1.2528 \cdot 10^{-4}$	$2.2907 \cdot 10^{-5}$
L3 hits	$-7.878 \cdot 10^{-5}$	$1.4998 \cdot 10^{-5}$
Bytes read from memory controller	$-5.8589 \cdot 10^{-8}$	$9.7646 \cdot 10^{-9}$
Bytes written to memory controller	$-3.8536 \cdot 10^{-8}$	$2.236 \cdot 10^{-8}$
Instructions retired	$1.184 \cdot 10^{-8}$	$2.744 \cdot 10^{-9}$
Interrupts	$3.659 \cdot 10^{-3}$	$1.3766 \cdot 10^{-3}$
Context switches	$-1.7381 \cdot 10^{-3}$	$5.0857 \cdot 10^{-4}$

Table 4.4.: *Estimated coefficients β of linear regression model*

From the observations the design matrix $\boldsymbol{X}$, the vector $\boldsymbol{y}$ and $\boldsymbol{\epsilon}$ can be constructed as shown in Equation 4.6. x_{14} in the design matrix refers to the first observation and the fourth performance counter value. The linear regression model can now be written in the compact form $\boldsymbol{y} = \boldsymbol{X\beta} + \boldsymbol{\epsilon}$ where $\boldsymbol{\epsilon}$ are the independent distributed errors ϵ_i, such that $E(\epsilon_i) = 0$ and $Var(\epsilon_i) = \sigma^2$.

$$\boldsymbol{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} \qquad \boldsymbol{X} = \begin{pmatrix} 1 & x_{11} & \dots & x_{1k} \\ \vdots & \vdots & & \vdots \\ 1 & x_{n1} & \dots & x_{nk} \end{pmatrix} \qquad \boldsymbol{\epsilon} = \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{pmatrix} \tag{4.6}$$

Using the least square estimates from Equation 4.7, $\boldsymbol{\beta}$ can be estimated to form the final linear regression model with its coefficients shown in Table 4.4.

$$\boldsymbol{\beta} = \left(\boldsymbol{X'X}\right)^{-1} \boldsymbol{X'y} \tag{4.7}$$

5. Implementation

Some of the software used in the course of this thesis was not readily available for use, therefore several software components had to be implemented. A traffic generator has to be used for the reference measurements and no readily available application does fit the requirements. It therefore had to be self implemented. The architecture and configuration of the traffic generator is described, followed by an adaption of an existing code base for a DPI firewall that suits the packets generated. To model workloads using performance counter, several implementations to trigger counter events are implemented and evaluated on the basis of accuracy and side effects. The performance counter implementation also includes a kernel module to help improve results when triggering certain events. The resulting implementations are used to create a framework to accurately trigger performance counter events to model power consumption. The framework's design choices and integration into Chauffeur is explained in the last section.

5.1. Traffic Generator and Receiver

This section describes the implementation of the traffic generator and receiver program written in Java 1.8. Generator and receiver are only distinguished through the command line parameters that are read on program startup. First the architecture of the program itself is outlined and the generation of packets is explained, followed by the description of configuration parameters.

The requirements for the load generator are a simple configuration as well as the ability to generate a high number of packets per second to saturate an ethernet connection with a bandwidth of 1 GBit/s. A multithreaded application is preferred but not necessary. As the workload used is a VNF firewall inspecting UDP packets, both valid and malicious packets have to be generated at defined rates with delay between packets determined by a distribution function. The generator must support the UDP protocol to be compatible with the firewall. The receiver application must only be able to count arriving UDP packets on specified sockets.

A traffic generator was implemented as available solutions do not have the required features. As stated in [Ols05], the Linux kernel *pktgen* module offers a high speed packet generator that can use multiple threads. Yet each thread needs exclusive access to a network device. This would limit pktgen to a single thread as only one ethernet link to the SUT is available. While it is able to saturate line rates of 1 GBit/s with a single thread,

packet delay distribution in pktgen is hard to control for small delays. A second traffic generator considered is *Ostinato*, which is highly configurable and comes with a wide variety of protocols, including UDP. This variety makes Ostinato difficult to configure properly and achieve the required load levels. If Ostinato could saturate the available ethernet bandwidth could not be determined in the course of this thesis. It is therefore decided to implement a traffic generator suited best for the required features.

5.1.1. Architecture

The architecture for the traffic generator consists of five packages as can be seen in Figure 5.1. The main package `nfv.traffic` contains classes for controlling the program from the command line, managing sender threads and logging facilities. Package `nfv.traffic.config` is concerned with parsing and writing JavaScript Object Notation (JSON) configuration files. Distributions for package send intervals are contained in `distribution`. The last package `nfv.traffic.protocol` does contain the sender and receiver classes for package generation.

A logging facility is provided in the form of a console and file logger in package `nfv.logger`. They have been implemented as an extra project due to its supplemental nature. It can therefore be omitted after small code changes if no logging is required.

Command and control tasks are running as one thread. Each socket bound by the traffic generator uses its own thread. Packet generation can therefore distributed to multiple threads to achieve a high amount of packets per second. The logging facilities run in their own thread storing log messages in a blocking data structure before writing. This approach is chosen to keep interference with packet generation and control to a minimum if log messages are written to disk.

A packet generated by the application is shown in Figure 5.2. Each packet has at least the following payload data. First an integer determining if this packet is part of a measurement, followed by an identifier if the packet is valid or malicious, which is either *valid* or *malicious* in ASCII representation. Afterwards the total packet count is appended for testing the implementation. It is followed by a unique identifier of the socket which did generate the packet. The rest of the packets payload is filled with randomly generated data. The minimum size of a packet is 21 byte for valid packets and 25 byte for malicious packets. The maximum size is limited by the Maximum Transmission Unit (MTU).

`nfv.traffic`

The class `Controller` in this package is the main class starting and initializing the traffic generator. It uses a single instantiation of `Command` to print out information to the command line. It also reads commands from the command line supplied by either a user or script. The available commands are:

- Start / Stop of traffic generators bound to specific sockets.
- Reset packet counters on a socket.
- Add / Remove / Copy new traffic generator sockets.
- Changing configuration of existing sockets.
- Printing packet counter variables in human readable form or as a comma separated list for automated parsing.
- Load / Store configuration files in JSON format.

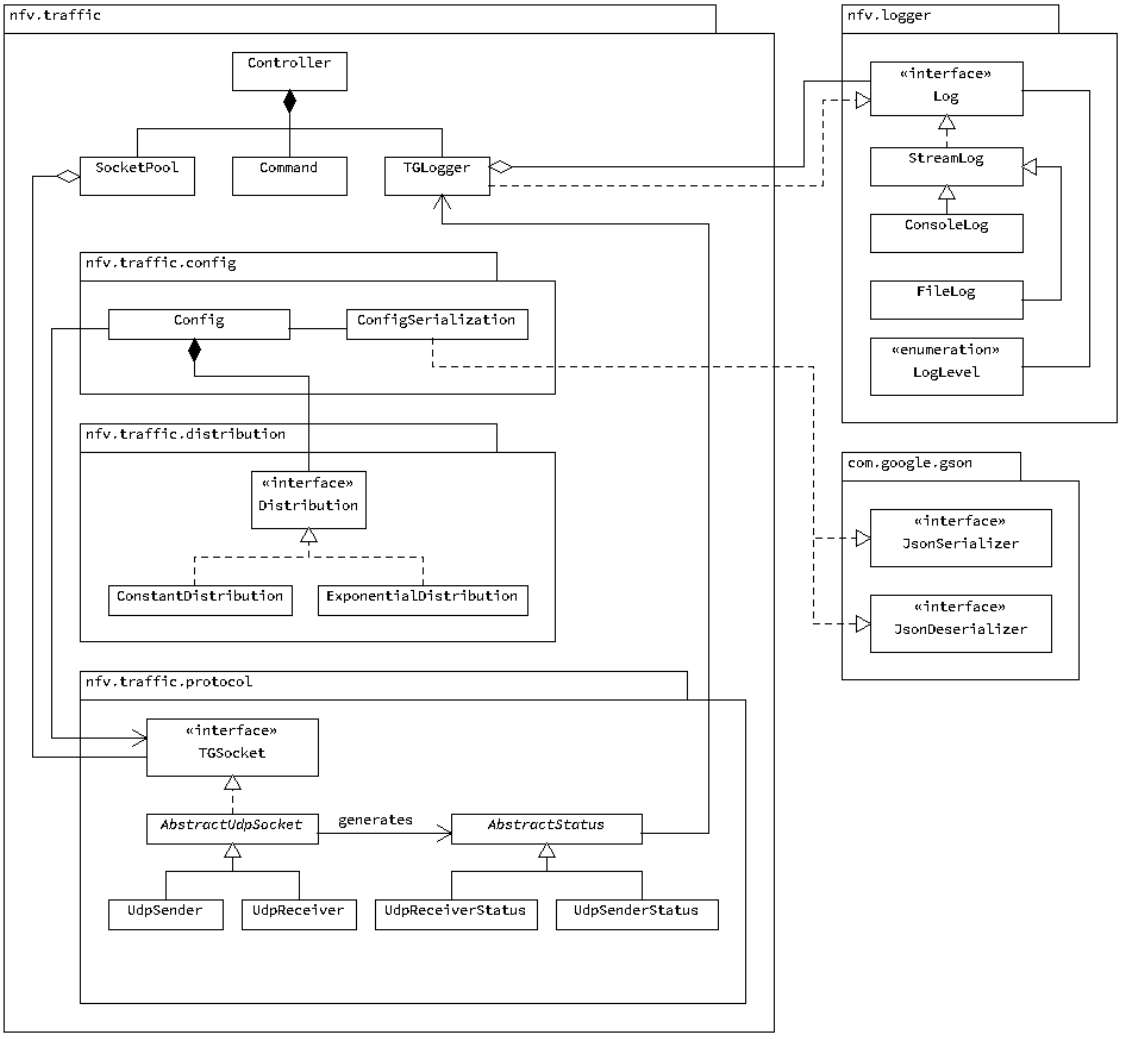


Figure 5.1.: *Software architecture of the traffic generator and receiver*

The `Controller` further includes an instance of `SocketPool` which is used for bookkeeping of bound sockets and associated threads. It is able to start, stop, add and remove packet generator threads from the application. The `Controller` object also contains `TGLogger` which keeps track of registered logging facilities.

`nfv.traffic.config`

The `config` package contains two classes. First the `ConfigSerialization` which parses and writes JSON configuration files by means of the Gson library. The second class in this package is `Config` which contains all settings needed for a traffic generator to run in unmodifiable fields. Immutability is chosen to prevent side effects on running packet generator threads by changing fields in a `Config` object.

Measurement identifier	Valid/Malicious identifier	Total packet count	Sender identifier	Random data
4 byte	5/9 byte	8 byte	4 byte	0 to MTU - 21/25 byte

Figure 5.2.: *Packet structure*

`nfv.traffic.distribution`

This package contains the `Distribution` interface consisting of just two methods. The first is to get a string representation of the distribution in use. The second method is generating the next delay according to the distribution implemented. It is based on the pseudo random number generator `Random` from the Java API. Two implementations for this interface are written. An exponential distribution (`ExponentialDistribution`) used for measurements and calibrations and a constant distribution (`ConstantDistribution`) for testing purposes.

`nfv.traffic.protocol`

This package contains most of the functionality for the packet generator. Each object of `UdpSender` and `UdpReceiver` is implemented as a thread bound to a socket. They are managed through the `SocketPool`. Each object counts how many packets were send in total, during a measurement phase as well as how many of these packets are malicious or valid. The current status of these counters can be returned as objects in the form of either `UdpSenderStatus` or `UdpRecevierStatus`. They are used by the `Command` class for printing. The configuration for a sender or receiver is determined during object construction by passing a `Config` object. The available configuration options are explained in detail in Section 5.1.2. Traffic generator threads are normally delayed using the `Thread.sleep()` method. An additional busy wait was added to be able to handle delays shorter than one millisecond to increase packet throughput and saturate the network link between the SUT and the traffic generator and receiver.

5.1.2. Configuration

This chapter describes the configuration options and their application in the traffic generator. The configuration file is divided into several blocks (objects in JSON), each consisting of the same 13 fields. For the receiver only the `localAddress` field is important. It is still necessary to always provide all fields or otherwise the file cannot be parsed correctly.

`localAddress`	The local address and source socket from which packets will be send. The socket has to be provided as a decimal Internet Protocol Version 4 (IPv4) address and port number divided by a colon.
`destAddress`	Destination address to which packets are send. The format is identical to `localAddress`.
`distribution`	The distribution function to be used. The value provided must match a string representation implemented by either `Constant` or `Exponential`.
`packetCount`	This value tells the traffic generator to stop after a certain amount of packets have been send on this socket. It is only taken into account if `usePacketCount` is set to `true`.
`packetSize`	This field determines the packet's payload size in bytes.
`usePacketCount`	If this field is set to `true`, the `packetCount` field is evaluated. If set to `false`, `packetCount` has no effect and the traffic generator will produce packets on this socket until stopped otherwise.
`meanDelay`	Mean value the distribution functions should achieve.
`minDelay`	Lowest value the distribution function should return.
`maxDelay`	Highest value the distribution function should return.

`maliciousRate` Rate of malicious packets generated.

`useStandardSeed` Set this value to `true` if the random number generator should use the standard seed implemented by Java. If set to `false`, the value `randomSeed` will be used to seed the random number generator.

`randomSeed` Seed for the random number generator. This value only takes effect if `useStandardSeed` is set to `false`.

`useBusyWait` Use the busy wait method instead of `Thread.sleep()` if set to `true`.

5.2. NFV Workload

The NFV workload running on the SUT is a DPI firewall implemented in C. The firewall was adapted to work with the packets generated by the traffic generator described in Section 5.1.1. It is able to distinguish between valid and malicious UDP packets depending on the payload.

The firewall uses the netfilterqueue library to receive packets from the Linux kernel. The library is used to create handles to which packets can be forwarded. The handle binds to a rule in the kernel netfilter module. To be able to bind, a netfilter rule needs to have a queue number assigned to it. If netfilter matches a packet to a bound rule, it is forwarding the packet to the handle. The firewall then has to render a judgment whether the packet should be dropped or accepted. If a packet is accepted, it is forwarded to the receiver. The firewall uses the string search function `strstr()` from the C standard libraries to search for either of the two identifiers described in Section 5.1.1. Does the packet include neither of the two identifiers, then the firewall will signal accept to netfilter. Packets not send over UDP will also be accepted.

Since the firewall is a single thread implementation, several instances have to be started and connected to rules in the netfilter kernel module. The rules for netfilter, configured through iptables, are explained in detail in Chapter 6.

The firewall uses only one command line parameter at startup. This is a positive integer number determining the queue number to which the firewall should be bound.

5.3. Linux Kernel Module

For the implementation of the performance counter framework, several implementations are tested. One of them is setting a contiguous memory range to not cache data and reliable trigger cache miss events. The module is written in American National Standards Institute (ANSI)-C as other languages are not supported. It creates devices and registers a callback function so user space programs can map virtual kernel memory to their memory space via a system call to `mmap()`.

5.3.1. Caching Modes

The different caching modes supported by the SUT's processor are described to gain an overview of which to choose. The used processor supports six different modes as stated in [Int16b].

Strong Uncachable (UC) All reads and writes will appear on the system bus and are not reordered. Also no speculative prefetches are made. This mode can only be set through the Memory Type Range Register (MTRR).

Uncachable- (UC-) This mode is identical to Strong Uncachable but can only be set through the Page Attribute Table (PAT). If set, it can be overridden if the same memory is set to Write Combining via the MTRR.

Write Combining (WC) Memory is not cached but writes can be delayed and stored in the WC buffer and written if the buffer is full or a serializing event occurs. Speculative memory access is allowed in this mode. WC allows reordering and is therefore for memory locations in which the write order is not important as long as the writes appear on the system bus. Cache coherency is not enforced by the CPU.

Write Through (WT) In this mode, read and writes are cached. Writes will be written to the main memory and cache line if it is valid. It allows for buffering as in WC and speculative memory access. Also cache coherency is enforced by the CPU.

Write Back (WB) This is the standard mode for main memory. All reads and writes are cached, speculative accesses can be made and coherency is enforced. Write misses will write to cache line and not to main memory until the cache line is evicted.

Write Protected (WP) In WP mode, read misses cause a cache line to be fetched. A write is is propagated to the system bus and the corresponding cache lines on all processors are invalidated. This mode allows for speculative read. It can only be set through the MTRR.

Caching mode Uncachable- (UC-) is selected as it can be set through the PAT and does not allow hardware prefetching of data. It also forbids the caching of data which in return should be a reliable way of triggering cache misses. Using the MTRR was refrained from in accordance with [GR], which states that the PAT should be used if available. Therefore Strong Uncachable (UC) is not selected. Write Back (WB) was not selected to allow for maximum control over how data is written or read without buffering. The modes Write Through (WT), WB and Write Protected (WP) are not suitable as they allow data caching. Furthermore setting memory as UC- has precedence over WB even if WB is set through the MTRR. This is important as most parts of the main memory are set to WB by default via the MTRR.

To set memory to UC-, access to the PAT is needed, which is only possible in ring 0. Therefore a Linux kernel module is written to allocate virtual kernel memory and mark it as UC-.

5.3.2. Character Device Driver

The module implements a character device driver. Devices in Linux use a major and a minor number. The major number decides which module in the kernel is responsible for the device. Choosing a hardcoded major number should be avoided as it could result in conflicts with other kernel modules using the same number. The preferred way for kernel modules is automatically assigning a free major number on module load. As a single module can manage multiple similar devices, each device also gets a minor number assigned. The minor number is used by the kernel module itself and not the kernel to distinguish multiple devices. In this module, the minor number is ascending, ranging from 0 to $n - 1$, with n as the number of devices that the module creates.

First sufficient space for the internal device representation is allocated and a major number is requested from the kernel. Afterwards a device class `uc` is created under which the uncachable devices will be grouped. This is not necessary but helps keeping an overview of all devices on the system. The group will show up as a folder in `/sys/devices/virtual/` containing all created devices assigned to this group. Afterwards the devices themselves are initialized. For each device, a certain number of contiguous pages in the kernel virtual memory space are allocated through `__get_free_pages()`. The returned pointer is used to set the allocated memory as UC- by calling `set_memory_uc()`. If the memory could be allocated and set to UC-, a device is created and registered at the kernel. After registering, the device becomes usable by user space programs. Accessing the device is still difficult because no handle in the `/dev/` folder is created automatically. This is done through a call to `device_create()` with the full name for each device. The name used for the devices are *uncachable-m* where m is the minor number of each device.

Only four operations are permitted to keep the code base minimal. Therefore only the following callback functions needed to be implemented.

`open` This function does set the status of a device to *open* and stores a pointer to the module's internal device representation in the field `private_data` of `struct file*`, supplied as argument by the kernel. This is important for access to the device when `mmap()` is called. This callback is executed if a device is opened via `open()` from user space.

`release` Resets the device state to *closed*.

`read` This callback function is implemented to be able to read the amount of allocated memory in bytes.

`mmap` This function is called if a user space program tries to map the device's memory into its virtual memory space. To map the correct memory for each opened device, the internal representation is retrieved from the `private_data` field and the page protection is updated through `pgprot_noncached()`. Afterwards a call to `remap_pfn_range()` maps the virtual kernel memory to user space.

If a function call should fail due to an error, a memory cleanup is performed if necessary and appropriate error messages are printed to the kernel ring buffer. The called function will then return with a negative integer value to indicate the error.

The kernel module supports two parameters that can be set on module load. It will fallback to default values if the parameters are not supplied.

`device_count` This parameter specifies the number of devices (n) that the module creates. The default value is 8.

`device_size` The size determines the amount of pages allocated in memory for each device. It must be provided as an exponent to base 2. A device size of 3 for example will result in 32 KiB of memory allocated for a page size of 4 KiB. The default value is 2.

5.4. Relevant Performance Counter

To model workloads on the basis of performance counters, it is necessary to reliable trigger events associated with the counters used in this thesis. As shown in Section 2.3, performance counter do have drawbacks such as over counting and non-determinism. Together with modern multithreaded preemptive operating systems, reliably triggering performance events becomes difficult due to shared resources, scheduling and other characteristics that

cannot be directly influenced without changing the underlying OS. Therefore different possible implementations had to be tested and measured to determine the most viable solution for certain counters, which is described in the following sections. The counter for L2 misses is omitted, as generating an L2 misses are implemented by producing L3 hits.

Cache Hit and Miss Triggers

Triggering cache hits and misses is essential for the implemented performance counter framework. To trigger cache misses and hits, an array is allocated that is at least twice the size of the L3 cache, which is 16 MiB for the SUT used in this thesis, according to [Int16a]. The program will initialize the cache by cycling through 8 MiB before any cache misses or hits are generated. This is to improve accuracy as, in an ideal scenario, the last cache size bytes should still reside in cache. For L3 cache events, the complete array is used. In case of L2 cache events, only a subsection with four times the size of the total L2 cache is used in the same array, always following p_3. Three pointers to the array are needed to determine which data was loaded last and should still reside in cache.

- p_3: Position of the last used data for L3 cache misses.
- p_{2c}: Position of the last used data for L2 cache misses or hits in the L2 subarray.
- p_{2l}: Position of the first byte in the L2 subarray.

For L3 cache events, the complete array is cycled in steps with a multiple of the cache line size (64 byte [Int16a]). Optionally a random factor can be added to test if hardware prefetching or other hardware features influences event triggering in the following sections. As the last L3 cache size bytes should still remain in cache after initialization, L2 hits are triggered first, followed by L3 hits and last L3 misses. Triggering L2 hits first ensures that only two fixed memory pointers should produce an acceptable hit rate on the first two bytes of the L2 subarray which are p_{2l} and p_{2c}. Afterwards the L3 hits are generated by cycling through an array four times the size of the L2 cache size (256 KiB per physical core [Int16a]) that still fits inside L3. This will move p_{2c} forward until it encounters the array's upper bound which will reset p_{2c} back to $p_{2l} + 1$. Finally L3 misses are produced by cycling through the complete array with p_3. Mixing different cache event triggers is possible by moving all pointers appropriately.

The cache hit and miss triggers are implemented with three different instructions to evaluate different approaches. Single Instruction Multiple Data (SIMD) intrinsics, C and x86-64-Assembler (ASM) is used to find the optimal implementation. Also included are different functionalities which are read, write and copy data. A random factor when stepping through the array is added to the pointer as well to evaluate influences of speculative access as described in 2.3. The memory used by the trigger implementation is either virtual process memory, shared memory or memory from the kernel module that is marked as UC-.

Read and Write Bytes to Memory Controller

To produce a memory access, all cache levels have to be missed beforehand. Therefore read bytes from memory controller and write bytes to memory controller are implemented in the same manner as the L3 cache misses. The memory controller counters are implemented with the three aforementioned variants, SIMD, C and ASM as well. The functionality on the other hand is restricted to the access type, which is either read or write. The implementation of the read and write event trigger also includes the three different memory types for evaluation.

Instructions Retired

Triggering events to modify the retired instruction count is possible with basic C or C++ code. Therefore a simple routine was implemented, adding a constant value to a stack variable as shown in Listing 5.1. Yet higher language constructs can consist of multiple instructions. The code is therefore compiled into ASM language as can be seen in Listing 5.2. The assembler instructions which comprise the loop construct, line 7 and 8 in Listing 5.1, can be found in line 27 to 33 and amounting to seven instructions per loop iteration. The remaining instructions are totaling 13. They could be subtracted from the number of events to generate, but it would only result in an immeasurable small improvement and is therefore not implemented. The Call Frame Information (CFI) directives are not counted as they do not get executed and are for stack unwinding in case of exception handling and debugging.

```
1 static uint64_t INST_RET_LOOP = 7;
2 static int32_t INST_RET_ADD = 5;
3
4 static void inst_ret_retire(uint64_t instruction_count) {
5         uint64_t loop_iterations = instruction_count / INST_RET_LOOP;
6         int32_t x = 0;
7         for (uint64_t i = 0; i < loop_iterations; i++)
8                 x += INST_RET_ADD;
9 }
```

Listing 5.1.: *C++ implementation of retired instructions*

Context Switches

The Linux OS keeps track of how many context switches have been performed during runtime. To trigger a context switch, either a new process or thread has to be created that can be switched. Another option would be to suspend the currently running modeled workload to signal the OS that it can switch in already existing processes or threads.

Suspending the workload to trigger context switches though has two disadvantages. The first reason to not further investigating this solution is the scheduling interval. The precision of the `sleep()`, `usleep()` and `nanosleep()` functions is dependent on the OS. On Linux, according to [Wol16], the standard precision is 1 ms which is insufficient if more than 1000 transactions per second have to be executed. Another reason is the possibility that no thread does have to be switched in during suspension. Especially under low load and with several CPUs available, it is not guaranteed that another thread is available to be switched in.

It was therefore chosen to create a thread with an empty workload that could be switched in and immediately finished for joining. To keep the compiler from removing an empty function due to optimizations, an inline assembler macro with no content was added as can be seen on line 4 in Listing 5.3. The number of context switches to be performed was divided by two as the newly created thread has to be switched in, and after finishing, the workload has to be switched back out for the transaction to resume.

Interrupts

Hardware interrupts are common on most systems and are being counted by the OS. They are selected as it is very likely that external traffic / load generators will cause interrupts on the SUT. Yet triggering a hardware interrupt programmatically is difficult without resorting to external hardware. One solution was found to achieve interrupts. Using the

```
_ZL13INST_RET_LOOP:
        .quad   7
        .align 4
        .type   _ZL12INST_RET_ADD, @object
        .size   _ZL12INST_RET_ADD, 4
_ZL12INST_RET_ADD:
        .long   5
        .text
        .type   _ZL15inst_ret_retirem, @function
_ZL15inst_ret_retirem:
.LFB0:
        .cfi_startproc
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        mov     rbp, rsp
        .cfi_def_cfa_register 6
        mov     QWORD PTR [rbp-40], rdi
        mov     rcx, QWORD PTR _ZL13INST_RET_LOOP[rip]
        mov     rax, QWORD PTR [rbp-40]
        mov     edx, 0
        div     rcx
        mov     QWORD PTR [rbp-8], rax
        mov     DWORD PTR [rbp-20], 0
        mov     QWORD PTR [rbp-16], 0
.L3:
        mov     rax, QWORD PTR [rbp-16]
        cmp     rax, QWORD PTR [rbp-8]
        jnb     .L4
        mov     eax, DWORD PTR _ZL12INST_RET_ADD[rip]
        add     DWORD PTR [rbp-20], eax
        add     QWORD PTR [rbp-16], 1
        jmp     .L3
.L4:
        nop
        pop     rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   _ZL15inst_ret_retirem, .-_ZL15inst_ret_retirem
        .globl   main
        .type   main, @function
```

Listing 5.2.: *ASM of retired instructions*

```
#define MIN_SWITCHES_PER_THREAD 2

static void context_op() {
        asm("");
}

void context_switch(uint64_t switches) {
        uint64_t switch_count = switches / MIN_SWITCHES_PER_THREAD;
        std::thread threads[switch_count];

        for (uint64_t i = 0; i < switch_count; i++)
                threads[i] = std::thread(context_op);
        for (uint64_t i = 0; i < switch_count; i++)
                threads[i].join();
}
```

Listing 5.3.: *Context switch implementation*

Boost open source library, they are triggered by programming the Advanced Programmable Interrupt Controller (APIC) to throw a local timer interrupt after a deadline is reached. The deadline is set to a value of 5 ns. This is the minimum value at which interrupts could be observed.

5.5. Performance Event Trigger Framework

Each workload generates performance events which are counted by the PMU. These counters can be leveraged using a generic approach. Performance Event Trigger Framework (PET) uses these counters to simulate workloads by triggering the same amount of counter events. To achieve a reliable generation of events, the implementations described in section 5.4 are used. PET can therefore be used to validate power models by simulating arbitrary workloads.

PET should be able to utilize only a subset of the implemented performance counters. It is therefore implemented as a modularized system to deploy only those event triggers needed for the modeled workload. Correctional factors and improvements presented in Section 5.4 are incorporated where feasible. The framework should also be able to add new performance counter triggers in the future without making major changes to the code base.

PET is primarily designed to work as a library for benchmark suites such as Chauffeur, but can be run as a standalone executable as well. If run as executable, a fixed amount of transactions supplied as a command line argument are performed.

5.5.1. Architecture

PET consists of three main parts as shown in Figure 5.3. Foremost the framework itself, keeping track of the modules, their operations available and the side effects imposed by the operations. The framework also includes a simulated annealing functionality to find a solution between the side effects and number of events to generate. The framework offers a native C++ interface and a Java Native Interface (JNI) for Chauffeur through an adapter. Lastly the modules triggering performance events are connected to the framework. The modules provide the event triggering implementations.

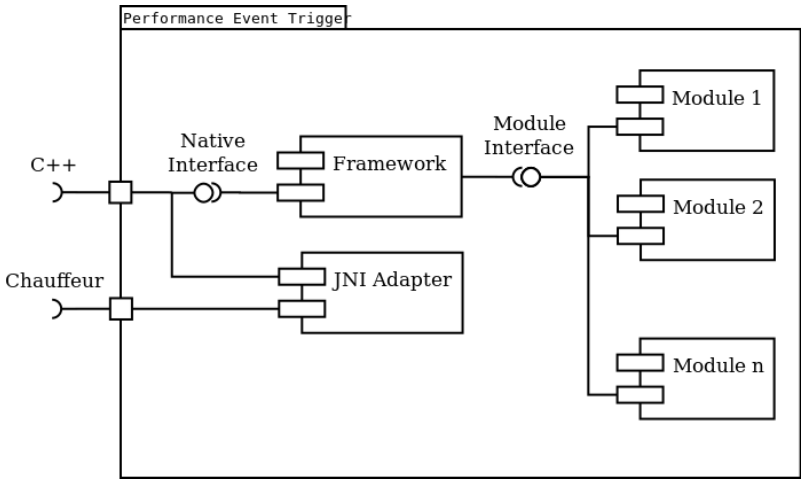


Figure 5.3.: *PET architecture*

Framework

The framework manages the modules that should be used, the operations that can be executed together with their respective values and the side effects. It takes care that all modules have been initialized and cleaned up via callback functions. The initialization and cleanup calls are shown in Figure 5.4 and 5.5. Internal housekeeping is omitted for clarity. PET first comprises a list of registered modules and available operations which is checked against the configuration. If a configured operation is not available, it is removed from the list. Is the simulated annealing flag is set to `true`, simulated annealing will be performed to approximate operation values with the configured values as target, taking into account the side effects the current configuration will impose. Afterwards the module list is iterated and each module's initialization function is called with the process number as parameter. If an error occurs during module initialization, an error code not equal to zero can be returned, resulting in the operation's values for this module to be zeroed. Afterwards the side effects are incorporated into the event values if the framework is set to use side effects, or the values from the simulated annealing are set. The original configured values are adopted unchanged, in case the naive approach without side effects is used.

Cleanup is a simple task of traversing the internal list of registered modules and calling the cleanup function on each. As with initialization, if an error occurs, it can be signaled by returning an error code. After cleaning up the modules, all values for a module's operation are set to zero.

Generating performance events is carried out by calling the `transaction()` function which subsequently will call the module's operation with the correct amount of events to be generated as shown in Figure 5.6. `init(int32_t)` must be called before `transaction()`. Otherwise the function will return immediately without generating events.

PET also supports simple debugging for the framework and modules. It provides the `debug(FILE*, const char*, ...)` function with the same declaration as `fprintf`. If the `DEBUG` flag is not set, calls to `debug` will have no effect. This keeps the code more readable as the debug output functions do not need to be removed if no debug is desired. It also removes pre processor `if-else` blocks which would otherwise be necessary. For all modules, the file identifier `FILE* debug_file` is available for the `debug` function. The framework ensures that the variable is initialized before any call to a module is made. Each process of PET writes to its own debug file. Therefore each invocation must be distinguishable. This is achieved by supplying the process number to the framework when

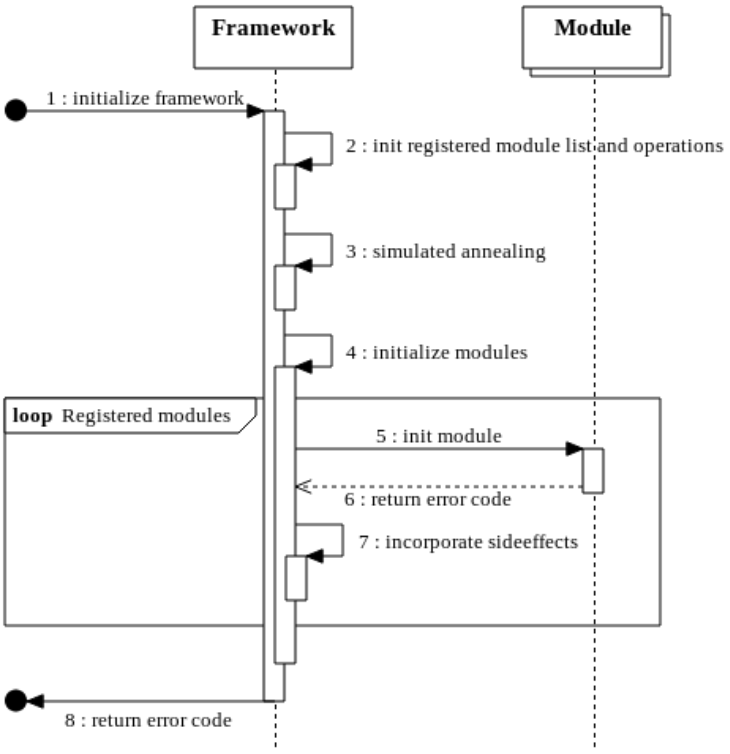


Figure 5.4.: *PET initialization*

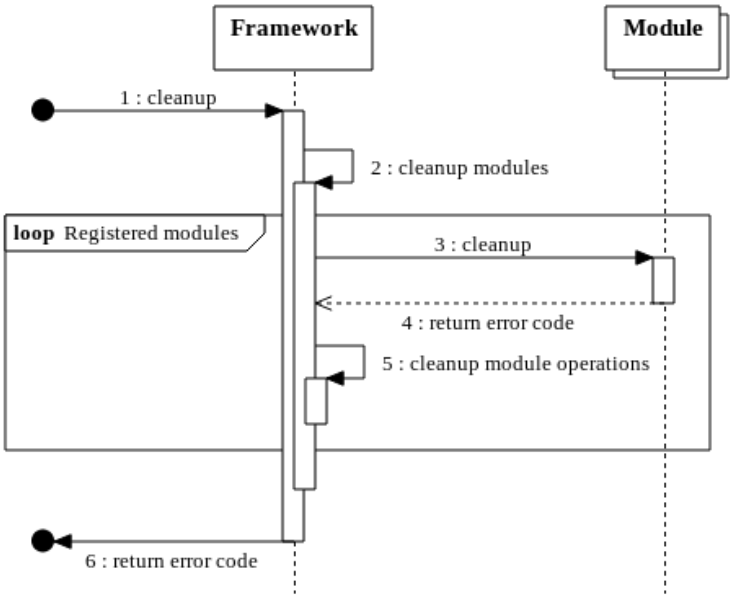


Figure 5.5.: *PET cleanup*

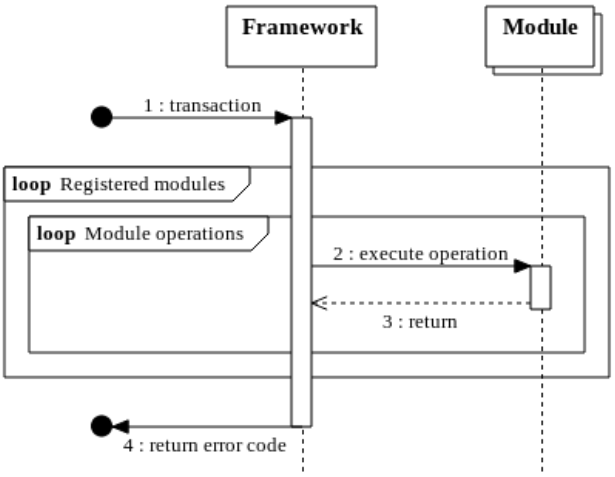


Figure 5.6.: *PET cleanup*

calling the `init` function. If PET is used as an executable, it must be provided as the first command line argument.

The PET library offers two interfaces, a native C++ interface and JNI to be called from a Java application. The interfaces should be easy to use and not overburdened. The native interfaces therefore consists of only eight functions while the JNI offers only three. The native interface functions in the `pet` namespace are:

- `int32_t init(int32_t)`
  Initializes PET so transactions can be executed. A call to this function will also cause all modules to be initialized via their respective initialization function. It will return a non-zero value if an error occurs. The only argument is the process number that must be supplied.
- `int32_t cleanup()`
  Resets PET, frees memory and calls the modules cleanup function. Will return a non-zero value if an error occurs.
- `int32_t transaction()`
  This function calls all performance event trigger operations with the supplied values in the configuration file.
- `std::list<pet::module*>* get_modules()`
  Returns a pointer to the list of modules configured.
- `void set_sideeffects(bool)`
  Switches the use of side effects on or off. Must be called before `init()` to take effect.
- `bool use_sideeffects()`
  Returns the status if side effects are used.
- `void set_simulated_annealing(bool)`
  Switches the use of simulated annealing on or off. Must be called before `init()` to take effect.
- `bool use_simulated_annealing()`
  Returns the status if simulated annealing should be used for the side effects.

The `init()`, `cleanup()` and `transaction()` functions are adapted to JNI to be callable from Chauffeur. Functions switching the side effects or simulated annealing on or off are not adapted and their values are provided as parameters to the JNI initialization function as shown below. The interface is automatically generated by `javah` and included in PET.

Modules

The modules provide the actual performance event triggering. A module offers one or more operations to trigger performance events. Calls to module operations should not fail or fail gracefully. Therefore no error code is returned from operation calls.

As the modules are build as static libraries, each module must have a unique name after which its callback functions are named to avoid errors during linking. All module functions and variables should be contained in the `pet_module` namespace to keep the global namespace uncluttered. To simplify the process of writing modules, C pre-processor macros have been implemented that provide appropriately named callback functions for the framework.

- `pet_module_register(name, init_func, cleanup_func)`
  This macro will create all necessary functions and data holding variables. They are named appropriately to be distinguishable for the linker. The module's name is the folder it resides in and is used in the configuration file. The name must be unique.
- `pet_add_operation(operation_name, callback_func)`
  This macro will add an operation to the list of available operations for this module. It must be called after `pet_module_register` to work properly. The name of the operation is also used in the configuration file for identification. The name must be unique within PET. An operation is called from PET via the callback function which must have the following function header format: `void callback_func(uint64_t)`.

The macros must be called within the `pet_module` namespace as shown in the example for the instructions retired module in Listing 5.4. In total four modules with the following operations are implemented.

- `cache_memory`
  - Generate L3 cache miss
  - Generate L3 cache hit
  - Read bytes from memory controller
  - Write bytes to memory controller
- `inst_retired`
  - Increase instructions retired counter
- `interrupt`
  - Generate hardware interrupts
- `ctxt_switch`
  - Trigger a context switch

5.5.2. Configuration

To configure the framework, a simple configuration file parser is implemented. The configuration file contains the modules to use, the amount of events to trigger for each operation and the side effects on other events caused by generating an event. A configuration file

```
#ifndef SRC_MODULES_INST_RETIRED_INST_RETIRED_H_
#define SRC_MODULES_INST_RETIRED_INST_RETIRED_H_

#include "../../include/module.h"

namespace pet_module {

// Instructions executed for each loop iteration
static const uint64_t INST_RET_LOOP = 7;
// Value added to the temporary variable to have an instruction that gets executed.
// This value is not relevant but magic numbers are bad coding style.
static const int32_t INST_RET_ADD = 5;

static void inst_ret_retire(uint64_t);

static int32_t inst_ret_init(const int32_t);
static int32_t inst_ret_cleanup();

pet_register_module(inst_retired, inst_ret_init, inst_ret_cleanup);
pet_add_operation(inst, inst_ret_retire);

};  // namespace pet_module

#endif  // SRC_MODULES_INST_RETIRED_INST_RETIRED_H_
```

Listing 5.4.: *Module header for generating retired instructions events*

is parsed during compile time, generating a header file populating internal lists for the modules, operations and side effects. Parsing is done via a Python script that is called from the build system as a dependency to the framework target.

The configuration file consists of three sections which must be in the correct order to work correctly as illustrated in Listing 5.5. Each section name is preceded by a `#` character. The first section, `MODULES`, includes all modules that should be used by the framework. The name of the module must be identical to the folder in which its source code resides in the PET subfolder `modules`. The second section, `OPERATION_VALUES`, lists the operations which should be performed and how many events should be generated for each transaction. If an operation name is supplied that is not offered by a listed module, it is discarded. If the same operation name is listed twice, the last occurrence will have precedence. The last section named `OPERATION_SIDEEFFECTS` describes the side effects an operation produces per event. The operation name is followed by a list of affected operations and by which factor each operation is affected. Values for side effects must be `double` values. All values provided are inserted in the source code. Therefore values not valid by the C++11 standard will lead to compiler errors. An example configuration file is provided together with PET.

Side Effects

The side effects supplied are measured as shown by running each performance event trigger on its own and record the values for the other counters. The recorded side effect values are divided by the total amount of events triggered to reflect the impact on other counters for a single performance event. If the side effects should be used via the `set_sideeffects` function, it can be switched between two different approaches, described in detail in Section 4.2.2.

First, a simple summation of all side effect factors multiplied with the causing event count configured. The sum is then subtracted from the affected event count. Cases in which an

```
#MODULES
cache_memory
inst_retired

#OPERATION_VALUES
l3miss 100
mem_read 500
inst 1000000

#OPERATION_SIDEEFFECTS
l3miss mem_read,factor,0.1;inst,factor,50.0f
mem_read l3miss,factor,4.2e-3;inst,factor,24
```

Listing 5.5.: *Example PET configuration file*

operation value would fall below zero are handled by setting the value to zero and therefore not executing the operation.

The second option is simulated annealing, which is offered by the external GNU Scientific Library (GSL) library. As energy function, a modified MSE function is used. The distance function is defined as the energy difference between two neighboring solutions.

If simulated annealing or the summation should be used can be switched by using the function `set_simulated_annealing`.

5.5.3. PET Build System

PET uses Cmake as build system to generate Makefiles. Cmake is chosen as it is independent of the operating system and offers a simple way of configuring dependencies. It also offers setting parameters before building a project to influence the build process. All modules residing in the folder `src/module` will be added to the build process if a `CMakeLists.txt` is provided. Only two commands are needed to build PET. The first command "`cmake .`" checks all dependencies, orders the targets accordingly and generates Makefiles. The second command "`make`" builds the project in the sequence determined by Cmake.

The framework build can be modified with the following five options configured through Cmake. The standard value for binary options is *off* if not otherwise stated.

debug_out This option switches debug output on or off. If set to *on*, the define `DEBUG` will be added to the compiler call, activating debug output as described in Section 5.5.1.

sideeffects Switches side effects on or off.

simulated_annealing Switches simulated annealing on or off. This option only takes effect if the side effects option is set to *on*.

config_file The PET configuration to use during compilation. The standard value is `../pet.cfg`.

classpath As the framework needs an adapter to be used from Chauffeur, class files for Chauffeur and the workload have to be set correctly. The standard setting are relative paths from the main PET folder, `../../bin:../../../Chauffeur/bin`.

Dependencies

The dependencies of PET are shown in Figure 5.7. Targets copying libraries and header files are omitted for clarity. To avoid version mismatches between libraries of the build system and the SUT, external shared libraries are searched first in the folder the framework is executed in. Therefore all external libraries needed for execution are copied to the bin folder. Libraries needed for execution are:

- `libboost_system`
- `libboost_date_time`
- `libstdc++`
- `libgsl`
- `libgslcblas`

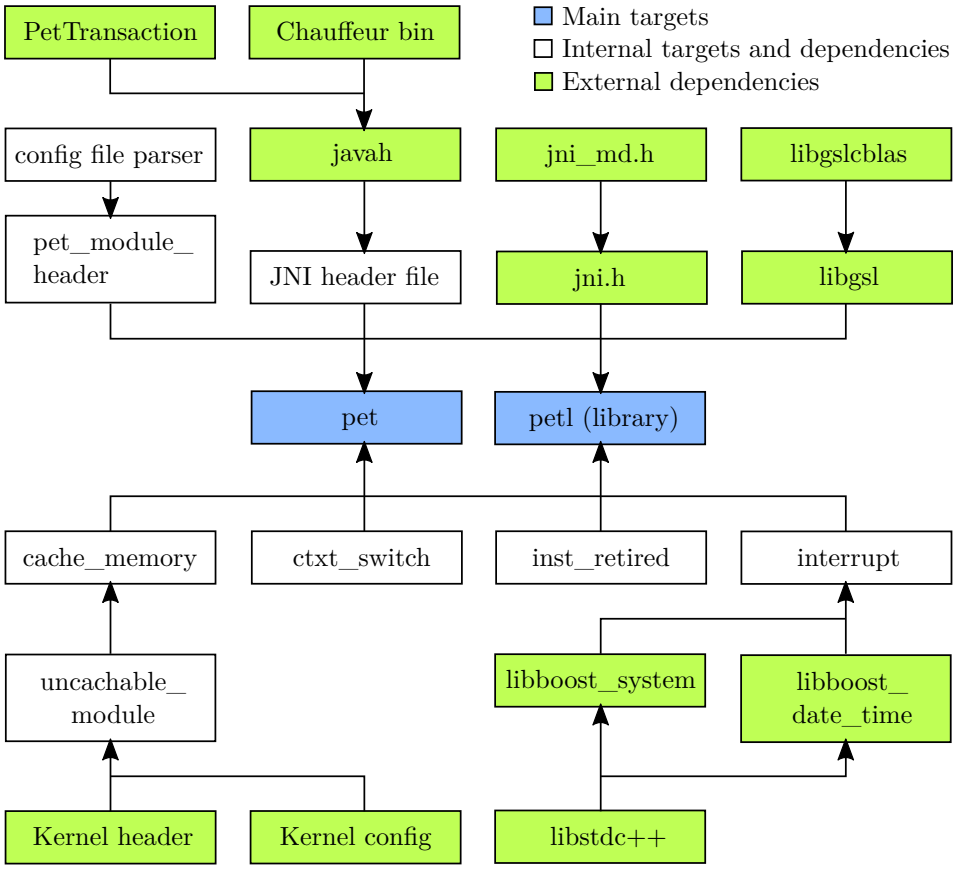


Figure 5.7.: *PET dependencies*

The JNI header file is generated through a call to `javah`. To correctly generate the header file, it is necessary to set the class paths in which the binaries of Chauffeur and the `PetTransaction` class can be found. They can be set through the *classpath* option. The `jni.h` and `jni_md.h` is included in the framework via the generated JNI header file. The configuration file parser generates a header file as well that is included by the framework. It contains the function to fill the internal module, operation and side effect lists with the parsed data. All files that need to be generated are created in the `bin` folder. This helps keeping the actual source code folder clean.

While most dependencies can be easily installed and deployed together with PET, the kernel module has a drawback. To build the kernel module, the correct kernel headers

and kernel configuration of the SUT is needed. It would be necessary to automatically collect the correct kernel information from the SUT to cross-compile the module. This was not within the scope of this thesis and the module needs to be build on the SUT to work correctly. A Makefile is supplied with the module. If it is not build on the SUT, then the kernel might refuse loading the module, panic and halt the system or crash without warning.

5.5.4. Chauffeur Worklet

A Chauffeur worklet is provided together with PET. The class `PetTransaction`, extending `Transaction`, contains all functionality. It has a static initializer block to ensure the native library is loaded before any instance is constructed. The overloaded `init(IntervalContext)` method provides the process number in form of the client identifier and forwards the call to the native functions. The `process(PetUser, Integer)` and `cleanup()` methods are forwarding the calls to the native functions as well without providing any parameters. Occurring errors during any of the three aforementioned methods are logged.

As no parameters have to be generated for a transaction, the `PetUserFactory` and `PetUser` are extending the Chauffeur base classes `AbstractUserFactory` and `AbstractUser`. They do not provide extended functionality. The `PetSuite` class is extending the Chauffeur class `BasicSuiteDescription` without functionality.

The interface `IPetConstants` provides the library path and name. The constants are used in `PetTransaction` to avoid cluttering the source code with string literals.

6. Testbed Setup

Two testbeds are used for this thesis. A reference testbed for measuring the VNF workload with the traffic generator and a second simplified testbed for evaluation measurements and the local workloads, Pi, XMLValidate and SSJ. This chapter describes both testbeds and their hardware configuration. The calibration for the VNF workload is also presented here. Calibration runs for the proposed framework and local workloads is done automatically with Chauffeur.

6.1. Reference Testbed and Calibration

The VNF reference workload needs a traffic generator to be stressed to certain load levels. Therefore a more complex testbed setup is necessary. It is illustrated in Figure 6.1 and consists of four machines.

The Controller System which runs the Controller, a self written script, calibrates the SUT and runs it at the desired load levels and collecting measurement data. It is connected to the Package Generator and Package Receiver, each running on a distinct system. The CPU and memory setup for the Generator and Receiver machines can be found in Section B.2. They report the number of malicious and valid packages generated / received during each run. For the power measurements, the Controller is also connected to the SPEC PTDaemon, which is collecting the power measurements from the power meter. The PTDaemon is running on the Controller System. The SUT is connected to the Controller as well and reports network statistics of received and send packets before and after each measurement phase. Performance counters are also measured on the SUT and reported back to the Controller. All control communication between the machines is transmitted via Transmission Control Protocol (TCP). The Package Generator is not only connected to the Controller but also to the SUT via a 1 GBit/s ethernet connection which carries the payload the DPI has to inspect. If a package is not dropped, it is forwarded to the Package Receiver through a second 1 GBit/s connection.

The Generator uses 16 threads for package generation to stress the SUT's 8 logical cores uniformly, with two sender threads per logical core on the SUT. On the SUT, each package is handed over to a specific firewall process depending on the source Internet Protocol (IP) address. As the modification of the source address is not possible within Java and all sent packages would have the same source, netfilter rules are applied on the Load Generator that changes the source depending on the destination port number shown in Listing 6.1.

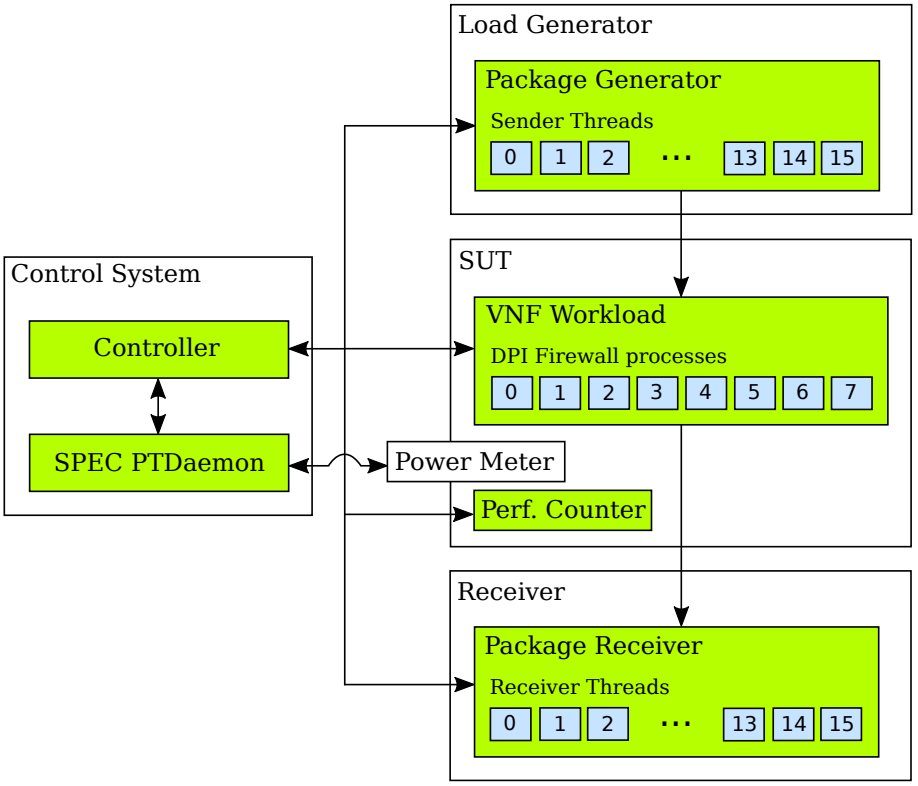


Figure 6.1.: *Setup of the reference testbed*

The `PORT` and `IP` variables are replaced accordingly. The rule is added to the NAT table to be able to change the source address after the package is routed to the receiver with the address `172.16.3.102`.

```
1 iptables -t nat
2          -A POSTROUTING
3          -d 172.16.3.102
4          -p udp
5          --destination-port ${PORT}
6          -j SNAT
7          --to-source 172.16.3.${IP}
```

Listing 6.1.: *Netfilter rule for the Load Generator*

The incoming packages need to be directed to the correct DPI processes running in the user space. As before, netfilter rules are applied that redirect the packages. The rule creation is shown in Listing 6.2. The rule states that all UDP packages with the specified source address (`-s`) and destination `172.16.3.102` are handed over to the `NFQUEUE` chain with a specific number (`COUNT`) ranging from 0 to 7, corresponding to a DPI process. The `IP` variable is identical on both the Load Generator and the SUT.

To distribute the load between all CPUs uniformly, each of the NIC's four RX interrupts is pinned to a physical core. As the receiver uses more, but less powerful cores, interrupts are also pinned to multiple physical cores on the receiver to avoid package drop between him and the SUT in case a single core cannot handle all arriving packages. The Package Receiver uses 16 threads, one for each possible source address.

```
iptables -A FORWARD
         -p udp
         -s 172.16.3.${IP}
         -d 172.16.3.102
         -j NFQUEUE
         --queue-num ${COUNT}
```

Listing 6.2.: *Netfilter rule for the SUT*

Calibration

After the testbed is setup several measurements with a mean send interval ranging from 0 to 18 000 ns per generator thread are taken to validate that the testbed is working. The results are shown in Figure 6.2. It can be seen that the implemented traffic generator is able to saturate the network connection. The packages generated in total are increasing while the TX queue stays constant at around 9000 ns. At the same time the send buffer errors rise with an identical rate as the generated packages, indicating that the NIC is under maximum load.

Despite testing with different queue lengths and settings for Generic Segmentation Offload (GSO) (hardware offloading is not supported on the SUT), there is a gap between the *Load Generator TX* and *Receiver RX* package counts for low mean send intervals. This package drop must occur on the SUT, as RX and TX queue values for the SUT are identical to *Load Generator TX* and *Receiver RX* respectively. It is also not reflected in the package drop counters of the NIC, but it will influence the *Valid Packages Received* used for calibration and is therefore calibrated out.

Calibrating the DPI firewall is performed via a binary search with a maximum of 20 steps. The maximum allowed package drop between the *Valid Packages Generated* and *Valid Packages Received* was set to 0.5 %. Three calibration runs were performed and resulted in an average send interval of 13 717 ns per thread as shown in Figure 6.2.

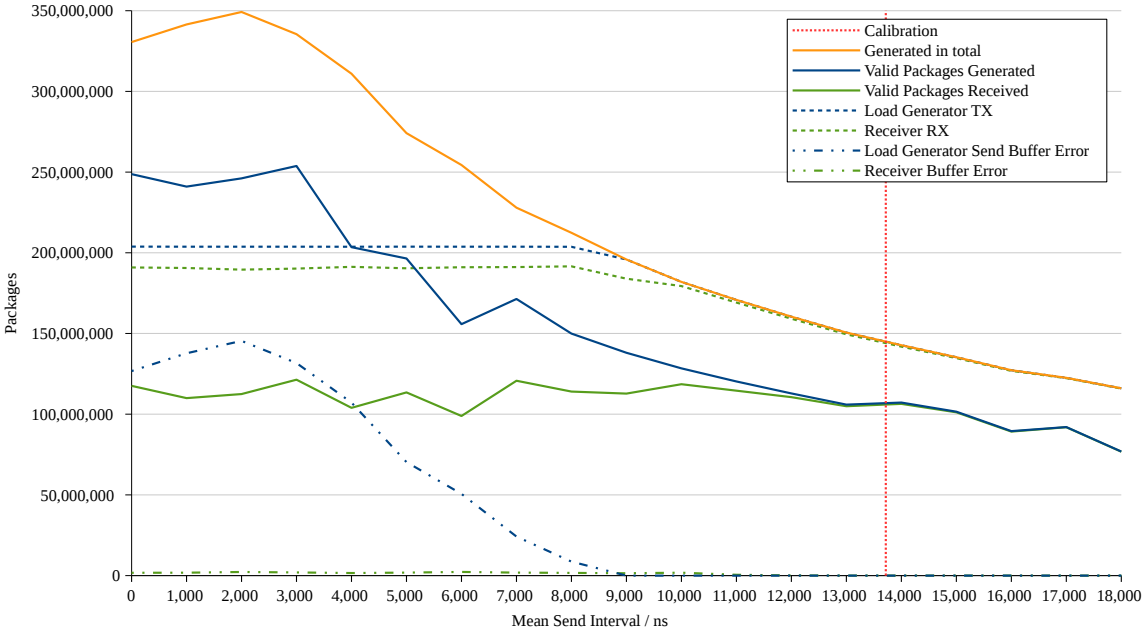


Figure 6.2.: *Calibrating the reference DPI firewall*

6.2. Chauffeur Testbed

Simplifying the testbed to be able to use Chauffeur, generally the same setup as shown in Section 2.2.2 is used with added measurements for the performance counters. It only consists of two physical machines as shown in Figure 6.3, the Control System and the SUT. The Control System is running the SPEC PTDaemon collecting power measurements and the Chauffeur Controller itself, which governs the SUT. The SUT reports back the measurement results and performance counters. It runs the Chauffeur Host, which communicates via TCP to the client processes executing transactions and IntelPCM for the counter values, described in Section 2.3. Each client is pinned to a single CPU core. The parameters for the simulated annealing library are shown in Listing 6.3. If it is used, the optimization will take place before every measurement during the initialization.

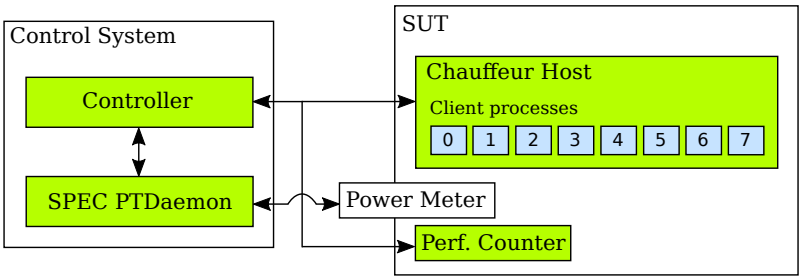


Figure 6.3.: *Setup of the simplified testbed*

```
static gsl_siman_params_t siman_config = {
        200,     // Number of points to try before stepping
        1000,    // Iterations before T is decreased
        2.0,     // Maximum step size
        1.0,     // Boltzman constant tk
        0.008,   // Initial temperature
        1.003,   // Temperature dampening factor
        2.0e-6   // Minimum temperature
};
```

Listing 6.3.: *Simulated annealing configuration of the GSL library*

7. Evaluation

The externally driven workload, is measured with power consumption and performance counters. Afterwards the framework is configured to approximate the NFV workload and measured on a simplified testbed with identical load levels. It is evaluated if the framework can approximate externally driven workloads and how accurate the approximation is. This is important as externally driven workloads can stress hardware that would otherwise remain unused for local workloads. The measurement results for the approximation is compared to the local approximation measurements to identify if hardware used only by externally driven workloads has an influence on power consumption and to which degree. To further validate the approximation, a regression model is build and its prediction is compared to the measured approximation results.

7.1. Performance Counter Implementation

To model workloads on the basis of performance counters, it is necessary to reliable trigger events associated with the counters used in this thesis, different possible implementations are tested and measured to determine the most viable solution for the event triggers. The evaluation for the event triggers is described in the following sections. The side effects for each event are determined for inclusion in PET. The event trigger for L2 misses is omitted, as generating L2 misses are implemented by producing L3 hits.

Evaluation of Implementation

The counters used include both hardware counters from IntelPCM (see Section 2.3) and software counters managed by the OS. For measuring how well the implementation performs, a measurement of 10 000 transactions is executed. The accuracy of events triggered is described in the corresponding sections. Also possible side effects on other performance counters are observed. They are evaluated and compared to the background noise by calculating the average number of side effects generated per transaction with the runtime and events generated in total.

The measurements for the performance counters are first run with one process. Followed by measurements with four and eight processes, number of physical CPUs and virtual CPUs of the SUT respectively, to test for their behavior in a multithreaded environment and determine the optimal solution for triggering events. To avoid possible deviations of events counted on a per core basis by changing the executing core for a process during runtime, each process is pinned to a fixed CPU number.

It is necessary for some counters to access large arrays residing in main memory to work. The following selected counter events are relying on this array to be generated.

- L3 cache misses
- L3 cache hits
- L2 cache misses
- L2 cache hits
- Bytes read from memory controller
- Bytes written to memory controller

Different memory sizes are measured for the aforementioned counters. As the memory is limited, allocation is dependent on the number of processes executed. For each step the memory size allocated by a single process is doubled from the last one measured, until the upper bound is exceeded. Measurement always begins at 16 MiB. For a single process, memory sizes up to 2048 MiB are deemed sufficiently large. For four and eight processes, upper bounds of 1024 MiB and 512 MiB are set respectively, resulting in a total of 4 GiB allocated.

SUT Background Noise

The measurements are not the only applications executing on the SUT. The OS and other background tasks still have to be processed on an idle system. These background applications might deviate counter values as they trigger events on their own. To account for background noise, a measurement of 120 s in an idle state is taken. The averaged values can be found in Table A.1. If events are counted on a per core basis, all cores are summed. The mean value is calculated for events counted on a per socket or system wide basis. For the evaluation of measurements, background noise is not removed as the target values are chosen to be high enough that the background applications are neglectable. The target values are two to four orders of magnitude larger than the background noise, depending on the performance counter.

7.1.1. L3 Cache Misses

Several different implementations and parameters for reliable triggering L3 cache misses are implemented to determine the optimal solution. The implementations differ in the instructions used to produce cache misses, which are SIMD intrinsics, C and ASM. Each instruction set was tested with read, write and copy functionality. For all options, the array is cycled through with a step size that is a multiple of the cache line size. The step size is varied with 2, 4 and 6 times the cache line size. For each transaction, 100 L3 cache misses should be generated.

To account for hardware prefetching used in modern CPUs, measurements that additionally add a random number in the range of 0 to 64 to the step size are taken. This applies only to virtual process and shared memory described below.

Another influence that is taken into consideration is the displacement of data already in the cache level by other processes. This could trigger unwanted cache hits or misses. Therefore the memory location in which the array is stored is measured with process owned virtual memory space and memory shared by all processes. In the case of shared memory, all processes are accessing the identical array. Also a third memory option is evaluated in the form of memory set to UC- through a kernel module as described in Section 5.3. Only one fixed array location is used for read or write operations and two if copy operations are executed. The locations for UC- memory are fixed as the data should not be cached by the processor and all read and write operations will appear on the system bus.

Singlethreaded

The evaluation for a single process without randomness on process virtual memory is shown in Figure 7.1. It can be seen that all implementations do not perform well and are not reaching the target of $1 \cdot 10^6$ L3 cache misses. The write functions in particular do generate almost no cache misses. One exception is the ASM read function that has only a deviation of less than $-3.5\,\%$ for memory sizes larger than 512 MiB with a step size of 6.

Measurement results if a random factor is used can be seen in Figure 7.2. In comparison without a random factor, an overall improvement can be observed. For a small step size of 2, all implementations fall short by a large margin. Increasing the step size to 4 and 6 improves the obtained results with $-6.6\,\%$ and $-3.5\,\%$ deviation for the ASM read function and a memory size greater 512 MiB. It can be seen that SIMD implementations do work but deviate further from the target value than the C and ASM implementations. Adding a random factor does not have an effect on write functions.

Using shared memory instead of virtual process memory did yield similar behavior for a single process as shown in Figure C.1 and C.2. It shows an unexpected behavior for the ASM copy function with step size 2, dropping sharply from 64 MiB to 128 MiB. As the improvement is not sufficient and it exhibits an odd behavior for different memory sizes, using the ASM copy function is refrained from.

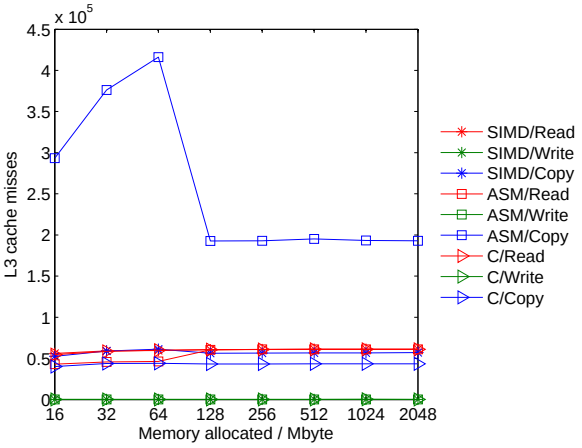
Setting parts of memory as UC- did not result in the expected improvement. Instead the opposite is the case. All implementations only produce a neglectable amount of L3 cache misses as shown in Figure 7.3. With an average runtime of 14.67 ms, the values are about ten times above the mean background noise of 117.6 L3 misses per second and core (see Table A.1). It can be deducted that a cache miss event is not triggered reliably, if the cache is set to UC-. An assumption is that the implementation itself causes some cache misses to be generated as overhead, but accessing UC- memory as intended is not counted towards cache misses.

Multithreaded

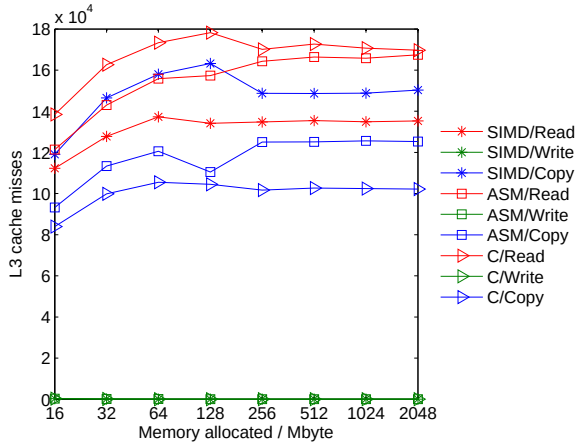
The implementations with or without randomness exhibits similar behavior as single process measurements for both, four and eight processes. Using a step size of 2 and 4 shows some odd behavior in the ASM copy function, as well as the C and ASM read function for eight processes (see Figure C.3 and C.8). These implementations do degrade with larger memory allocated without random factor. Using a random factor yields analogous results as single process measurements for process owned virtual memory as well as shared memory as shown in Figure C.4, C.6, C.9 and C.11. The kernel module providing uncachable memory did not perform better in a multithreaded scenario as can be seen in Figure C.7 and C.12. Therefore using the kernel module for producing cache misses was not further investigated.

As using shared memory is comparable to virtual process memory space and the implementation is requiring more complex system calls, it is also omitted in further cache miss evaluations. It can also be concluded that a step size of 6 works best overall and the C and ASM functions exhibit a higher accuracy than the SIMD function. The write functions are neglected as they did not perform as expected and produced only a small number of L3 cache misses. They will also inevitably introduce side effects on both, the bytes written to and bytes read from memory controller counter. The write functions did not perform better than the read implementation and are therefore not further investigated.

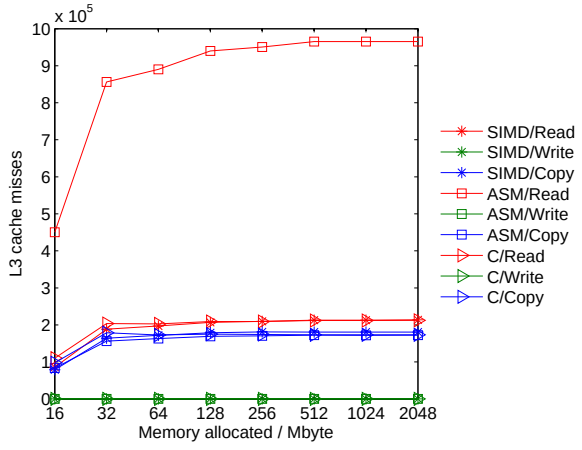
The ASM implementation does exhibit the most promising characteristics with and without randomness. No randomness is selected as it does not improve the ASM implementation. A memory size of 512 MiB is selected as a compromise between accuracy and memory footprint.



(a) Step size 2

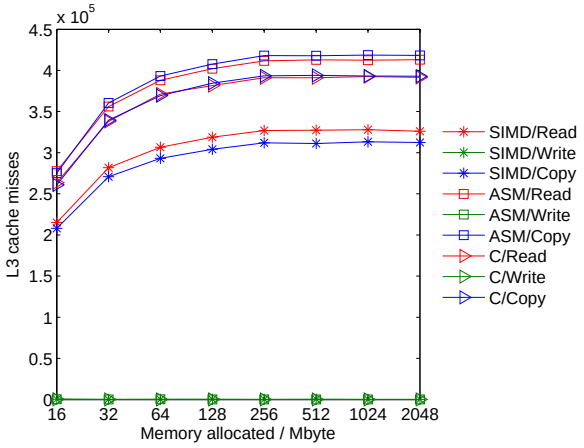


(b) Step size 4

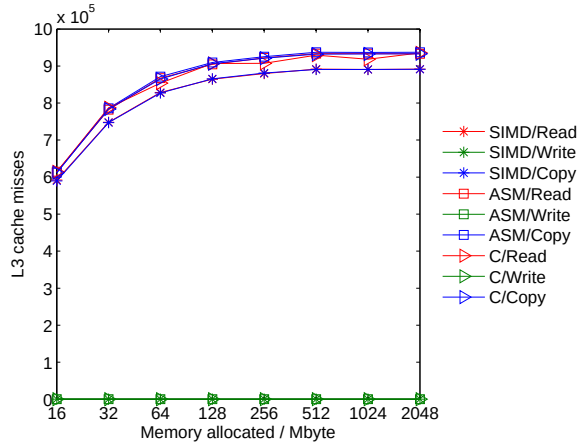


(c) Step size 6

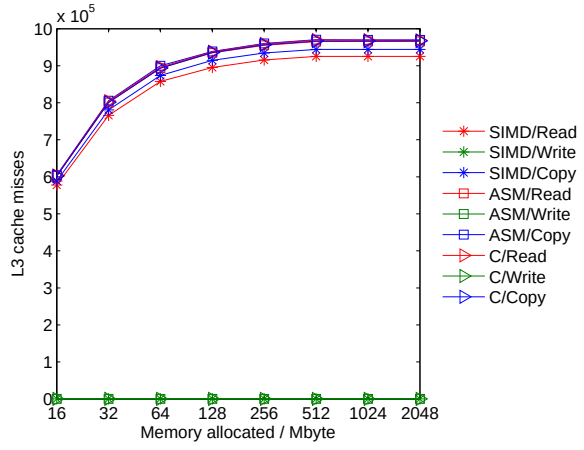
Figure 7.1.: *L3 cache miss results for virtual process memory and no random factor*



(a) Step size 2



(b) Step size 4



(c) Step size 6

Figure 7.2.: *L3 cache miss results for virtual process memory with random factor*

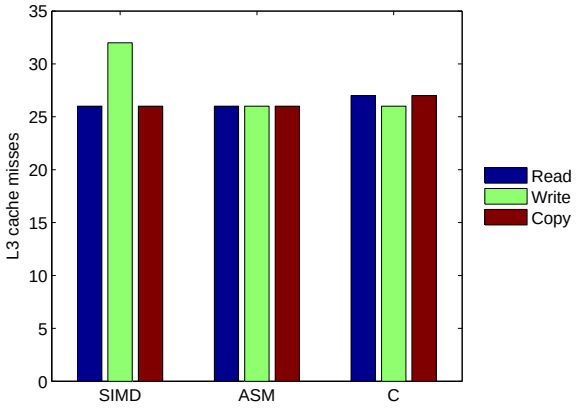


Figure 7.3.: *L3 cache miss results for UC- memory with fixed array indices*

Side Effects

The measurement results on other performance counters are shown in Table 7.1. While most counters exhibit the expected behavior with rising side effects when multiple processes are used, bytes read from memory controller is decreasing. It might be possible that the read counter converges towards the cache line size as more processes use memory bandwidth and speculative prefetches are decreasing. Yet the read counter decreases despite not fully utilizing the available memory bandwidth of 34.1 Gbyte s^{-1} [Int] with the highest total read/write transfer rate of 6.65 Gbyte s^{-1} for eight processes. All side effects, except L2 cache hits for a single process, are above the background noise.

	Processes		
Performance counter	1	4	8
L3 cache hits	0.0016	0.0025	0.0043
L2 cache hits	0.0018	0.0053	0.0231
Bytes read from memory controller	195.3955	98.5811	85.5179
Bytes written to memory controller	3.2293	4.6702	21.1981
Instructions retired	129.0967	129.6483	131.0951
Interrupts	$1.8 \cdot 10^{-5}$	$1.7 \cdot 10^{-5}$	$3.2 \cdot 10^{-5}$
Context switches	$17.3 \cdot 10^{-5}$	$13.9 \cdot 10^{-5}$	$22.0 \cdot 10^{-5}$
Average runtime (ms)	44	61.75	119.63

Table 7.1.: *Side effects of L3 cache misses per cache miss generated*

7.1.2. L3 Cache Hits and L2 cache misses

L3 cache hits are produced identical to L2 cache misses. This section therefore covers both. The cache hits are measured similar to L3 cache misses, except only a step size of 6 will be evaluated. A step size of 6 is chosen as it was shown in section 7.1.1 to have the most promising characteristics. Memory allocated is not evaluated for different allocation sizes as L3 hits are generated through a small subarray that fits inside L3 but not L2 cache. Therefore measurement results from 512 MiB are presented in accordance with the results shown in Section 7.1.1. The kernel module does not need to be measured as caching must not be disabled to generate cache hits. For each transaction, 100 cache hits should be generated, totaling $1 \cdot 10^6$.

Singlethreaded

The results for a single process with memory allocated from virtual process memory is presented in Table 7.2. Not applying a random factor while stepping through the array leads to fewer cache hits than targeted by a large margin. Using shared memory instead of virtual process memory does not show an overall improvement for all implementations. It can also be observed that the ASM and C functions perform better than SIMD. Results for the ASM copy function and the C read function are the most promising. As with L3 cache misses, it is refrained from using the copy implementation due to possible side effects that could increase both, the read and written byte counters without major improvements in accuracy.

		Virtual memory		Shared memory	
Implementation		No random	Random	No random	Random
SIMD	Read	165 031	923 975	204 095	921 490
	Write	932	1420	882	1501
	Copy	193 001	922 185	215 001	922 891
ASM	Read	204 282	966 181	215 075	967 565
	Write	914	1489	1273	1408
	Copy	222 718	979 248	217 607	965 657
C	Read	213 862	971 336	215 489	966 999
	Write	778	1421	960	1475
	Copy	259 416	968 232	265 018	966 856

Table 7.2.: *L3 cache hit results for a single process*

Multithreaded

Table 7.3 shows the results for four processes with a target value of $4 \cdot 10^6$ byte. With a deviation of $-11.5\,\%$, the C read function performs best on virtual process memory. The ASM and SIMD functions deviating further from the target value. Using shared memory yields no improvement over process memory.

		Virtual memory		Shared memory	
Implementation		No random	Random	No random	Random
SIMD	Read	443 037	3 391 786	1 193 440	3 110 869
	Write	6984	6088	8529	17 300
	Copy	1 328 717	2 218 435	872 237	3 305 744
ASM	Read	1 225 584	3 408 932	272 529	3 224 223
	Write	12 381	4740	10 786	14 939
	Copy	1 459 860	3 155 795	753 391	3 433 311
C	Read	743 568	3 537 446	727 382	3 279 071
	Write	8872	13 675	9379	9920
	Copy	1 358 422	2 701 179	2 120 709	3 354 666

Table 7.3.: *L3 cache hit results for four processes*

Results for eight processes are presented in Table 7.4. It shows major discrepancies compared to one and four processes. While the step from one to four processes scales reasonably

well, increasing the process count above the number of physical cores does not. A possible reason might be the L2 cache that must be shared between two processes for SMT. Exceptions are the SIMD read function on virtual memory and the ASM read function on shared memory. As the SIMD functions did not perform as well before, it is unreasonable to use this implementation for just eight processes. The C read shows the most deterministic behavior overall that scales with the process count up to the physical core count. While the ASM function performs well on shared memory on eight processes, due to the implementation using the same memory for L3 cache misses and hits, virtual process memory should be used for L3 hits as well. Allocating shared memory for L3 cache hits would make it necessary to produce L3 misses until the shared memory has replaced the virtual process memory used for L3 misses decreasing the accuracy. Therefore the C read function on virtual process memory is selected.

		Virtual memory		Shared memory	
Implementation		No random	Random	No random	Random
SIMD	Read	2 284 193	5 253 885	1 369 922	2 806 889
	Write	22 623	31 034	25 669	33 562
	Copy	1 579 026	3 738 683	1 293 117	4 273 203
ASM	Read	1 747 484	4 047 238	1 317 336	6 243 432
	Write	26 052	34 182	30 485	30 371
	Copy	2 478 220	4 343 272	1 418 783	3 881 193
C	Read	1 516 549	4 199 528	1 495 260	4 320 079
	Write	26 676	28 181	21 899	30 297
	Copy	1 556 921	4 320 763	1 973 445	3 612 386

Table 7.4.: *L3 cache hit results for eight processes*

Side Effects

The side effects on other performance counters are small but not neglectable as all side effects are above the background noise. Some odd behavior for the read can be observed while triggering this counter when using more than one process. While the value is low for a single process, the increase is large when multiple processes are executing L3 cache hits. Yet it does not increase when the process count is increased from four to eight. The L3 cache misses also exhibit a non-monotone behavior.

	Processes		
Performance counter	1	4	8
L3 cache misses	0.0026	0.0542	0.0389
L2 cache hits	0.0093	0.0120	0.0272
Bytes read from memory controller	0.5995	49.6423	38.4774
Bytes written to memory controller	0.4137	1.8019	10.4447
Instructions retired	74.2021	75.1916	75.7607
Interrupts	0.000 01	0.000 07	0.000 25
Context switches	0.000 17	0.000 56	0.000 71
Average runtime (ms)	13	46.75	97.75

Table 7.5.: *Side effects of L3 cache hits per cache hit generated*

7.1.3. L2 Cache Hits

L2 cache hits are measured with identical options as L3 cache misses in Section 7.1.1. Instead of traversing an array step-wise, fixed memory locations are used. This should keep the cache lines containing the addresses in cache and produce L2 hits. As the locations are fixed, no random factor needs to be considered. Also step size variations are not measured. The kernel module is not used as caching must be active to be able to hit the L2 cache. A target value of $1 \cdot 10^6$ L2 cache hits is set, with each transaction performing 100 hits.

Singlethreaded

The result for a single process generating L2 cache hits for virtual process memory are illustrated in Table 7.6. It can be seen that neither of the implementations is generating enough cache hits to reach the target value. The same applies when shared memory is used.

Implementation		Virt. memory	Shared memory
SIMD	Read	1601	1396
	Write	1557	1535
	Copy	1623	1606
ASM	Read	1392	1395
	Write	1512	1472
	Copy	1410	1394
C	Read	1502	1421
	Write	1378	1393
	Copy	1372	1506

Table 7.6.: *L2 cache hit results for a single process*

Multithreaded

L2 cache hits do not fare better in a multi process scenario. Table 7.7 shows that the L2 cache hits do increase. Yet they are still unable to reach the target values. It must therefore concluded that the technique and implementation of triggering L2 cache hit events is insufficient. The side effects are not evaluated as triggering does not work and this event triggering implementation should not be used.

7.1.4. Bytes Read from Memory Controller

This performance counter sums up bytes that have been read from main memory. Triggering this performance event is similar to that producing L3 cache misses as accessing main memory must miss all cache levels before. The test procedure is therefore identical to that described in Section 7.1.1 with the following two exceptions.

As shown for L3 cache misses, a step size of 6 performs best in triggering cache misses. Therefore other step sizes have not been examined. Only the read implementations are measured as a second change because copying could introduce major side effects. Writing on the other hand is assumed would not trigger the counter and is therefore also omitted. The implementations for the read function used are also identical to the ones described in Section 7.1.1.

Implementation		4 Processes		8 Processes	
		Virt. memory	Shared memory	Virt. memory	Shared memory
SIMD	Read	9467	8858	29 053	33 305
	Write	9965	10 398	33 351	32 110
	Copy	9093	10 385	35 694	28 268
ASM	Read	9493	9816	33 219	35 201
	Write	9648	9305	29 659	31 961
	Copy	9910	9061	32 164	31 329
C	Read	10 364	10 594	30 302	32 038
	Write	9110	7973	29 145	33 162
	Copy	9899	9487	27 992	32 317

Table 7.7.: *L2 cache hit results for four and eight processes*

Singlethreaded

First measurements for a single process are taken. While 100 byte are requested to be read for each of the 10 000 transactions, resulting in theoretical $1 \cdot 10^6$ byte in total. While a single byte will cause an entire cache line to be fetched, a target value of $64 \cdot 10^6$ byte is expected. Yet Figure 7.4 shows that each transaction generates between 13 350 byte to 14 285 byte of data read for the smallest memory size with even higher values for larger memory sizes. The overcounting decreases for larger memory sizes of 512 MiB and above for implementations without a random factor. The same behavior can be observed when using shared memory as shown in Figure C.15, which is due to the fact that only one process is generating read events. The results still show a more than two time overcounting on all implementations and memory sizes.

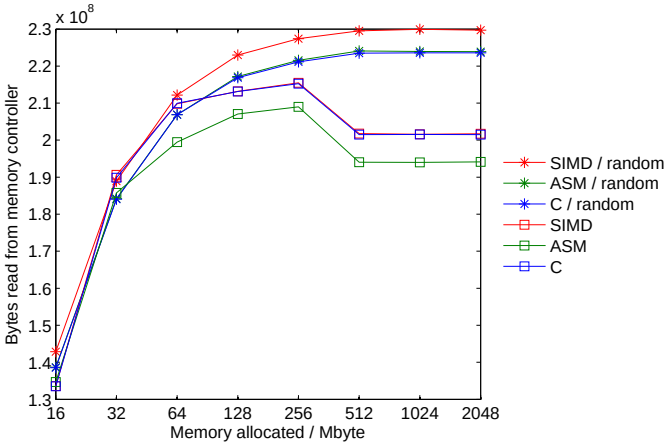


Figure 7.4.: *Bytes read results for single process with virtual process memory*

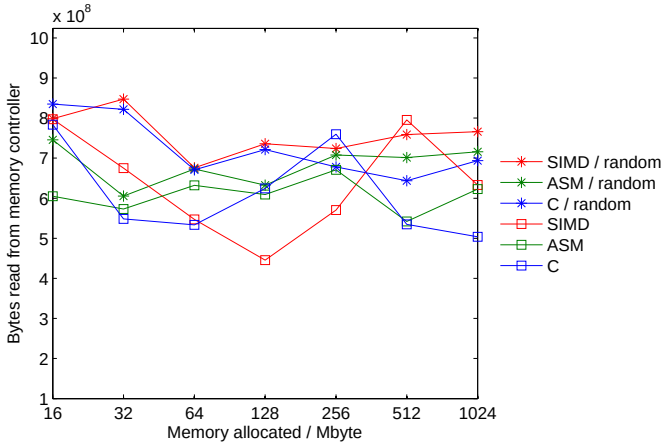
If the kernel module with uncachable memory is used, overcounting can be reduced to a minimum. While the module could not increase the accuracy in reliably triggering L3 cache misses, using it to minimize over-provisioning in read counters has proven effective, as Table 7.8 shows.

Implementation	Target	Bytes read	Deviation
SIMD	$64 \cdot 10^6$	64 002 560	0.004 %
ASM	$64 \cdot 10^6$	64 038 080	0.060 %
C	$64 \cdot 10^6$	64 025 024	0.039 %

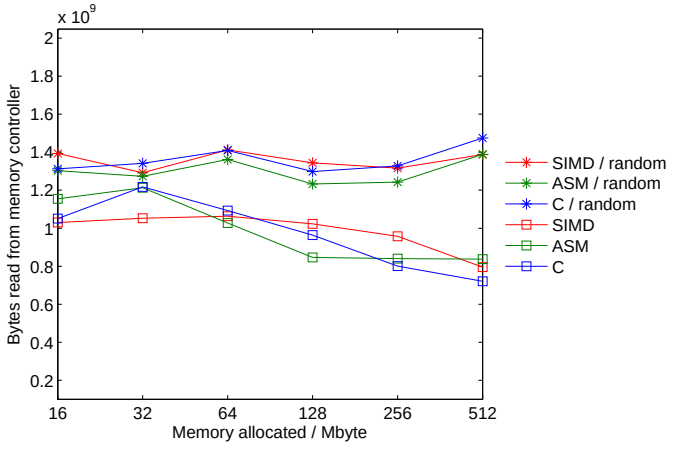
Table 7.8.: *Bytes read results for single process with uncachable memory*

Multithreaded

The results for multiple processes are shown in Figure 7.5. The deviations are ranging from 174 % to 331 % with four, and 141 % to 288 % with eight processes for target values of $256 \cdot 10^6$ byte and $512 \cdot 10^6$ byte respectively. Also large fluctuations across memory sizes and implementations make these counters less suitable if more than one process is running in parallel using virtual process memory. Using shared memory does not bring an overall improvement and is still largely overcounting as shown in Figure C.16.



(a) 4 processes



(b) 8 processes

Figure 7.5.: *Bytes read results for four and eight processes with virtual process memory*

Using the kernel module for multiple processes improves the results as shown in Table 7.9. Especially in the case of four processes, using the ASM read function comes close to the target value with a small deviation. With eight processes, deviations are larger with over

−30 % across all implementations.

Processes	Implementation	Target	Bytes read	Deviation
4	SIMD	$256 \cdot 10^6$	$242.41 \cdot 10^6$	−5.31 %
	ASM	$256 \cdot 10^6$	$255.98 \cdot 10^6$	−0.01 %
	C	$256 \cdot 10^6$	$191.16 \cdot 10^6$	−25.33 %
8	SIMD	$512 \cdot 10^6$	$326.43 \cdot 10^6$	−36.24 %
	ASM	$512 \cdot 10^6$	$343.67 \cdot 10^6$	−32.88 %
	C	$512 \cdot 10^6$	$347.82 \cdot 10^6$	−32.07 %

Table 7.9.: *Bytes read results for four processes with uncachable memory*

A simple correction factor for eight processes was tested. The results are shown in Table 7.10. Despite purposely choosing a lower value of 1.3 instead of 1.5, which is the average of all implementations, an overcounting can be observed. Yet the results show a lower deviation as before. It is questionable if the simple factor is sufficient as a high deviation only occurs for a CPU count that is above the number of physical CPUs. It is therefore specific to the SUT and a different approach might be more promising.

Implementation	Target	Bytes read	Deviation
SIMD	$512 \cdot 10^6$	$572.75 \cdot 10^6$	11.87 %
ASM	$512 \cdot 10^6$	$561.53 \cdot 10^6$	9.67 %
C	$512 \cdot 10^6$	$557.23 \cdot 10^6$	8.83 %

Table 7.10.: *Bytes read results for eight processes with correction factor and uncachable memory*

Reliable triggering read counters for a higher CPU count might not be feasible as the results for eight processes show. Still the ASM read function performs similar to the C function in this case and works well on lower process counts. The ASM read function in combination with the kernel module is determined to be the optimal solution to trigger the performance counter for bytes read from memory controller. The correction factor is not used as it is now overcounting on all implementations and the resulting framework should be agnostic to the amount of processes and the machine it is running on.

Side Effects

The side effects using the read event trigger show reasonable values for one and four processes in Table 7.11 and are all above the background noise. A small number of cache hits and misses are expected because it is both shared cache for instructions and data. Stepping from one to four processes yields only a smaller increases with the exception on the L3 cache misses, which increase significantly. Using eight processes a different characteristic is observed with a four to five times increase on the L3 and L2 cache counters and context switches. The bytes written counter especially is several orders of magnitude larger than for one and four processes. This strong incline in counted events, especially bytes written could be the reason for the implementation not performing well with higher process counts. Yet the underlying problem could not be determined within the scope of this thesis.

	Processes		
Performance counter	1	4	8
L3 cache misses	$2.7 \cdot 10^{-5}$	$9.3 \cdot 10^{-5}$	$42.5 \cdot 10^{-5}$
L3 cache hits	$7.180 \cdot 10^{-4}$	$8.898 \cdot 10^{-4}$	$36.074 \cdot 10^{-4}$
L2 cache hits	$1.551 \cdot 10^{-3}$	$2.703 \cdot 10^{-3}$	$8.417 \cdot 10^{-3}$
Bytes written to memory controller	0.0135	0.002 46	13.589
Instructions retired	22.014	22.163	24.078
Interrupts	$2.7 \cdot 10^{-5}$	$2.6 \cdot 10^{-5}$	$3.1 \cdot 10^{-5}$
Context switches	$1.74 \cdot 10^{-4}$	$1.58 \cdot 10^{-4}$	$8.51 \cdot 10^{-4}$
Average runtime (ms)	61	65.25	144.88

Table 7.11.: *Side effects of bytes read from memory controller per event generated*

7.1.5. Bytes Written to Memory Controller

This performance counter sums up the bytes written to the memory controller. The same test procedure is used as described for read bytes to memory controller in Section 7.1.4. The amount of bytes written per transaction is 100 byte totaling to $1 \cdot 10^6$ byte. As the caching mechanism will write complete cache lines instead of single values as WB (see Section 5.3) is set, a target value of $64 \cdot 10^6$ byte is expected to be measured.

Only write operations together with a step size of 6 are used to minimize side effects due to expected read operations when copying. Other step sizes are not evaluated as it has been shown in Section 7.1.1 that a size of 6 is optimal.

Singlethreaded

The results for a single process are presented in Figure 7.6 for virtual process memory. The implementation works well for memory sizes of 512 MiB and larger. With deviations with randomness ranging from 0.01 % to −0.03 % for the SIMD and ASM implementation respectively. Without random factor, deviations range from −0.03 % for SIMD to −0.01 % for ASM. For a memory size of 16 MiB not using a random factor is not optimal. Adding a random factor improves the results but still a large discrepancy can be observed. Using shared memory exhibits similar characteristics as can be seen in Figure C.17.

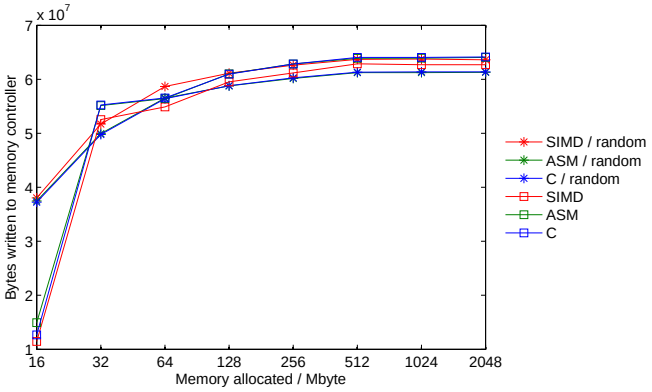


Figure 7.6.: *Bytes written results for single process with virtual process memory*

Using the kernel module as shown in Table 7.12 is expected to work well as observed before in Section 7.1.4 for the read bytes counter, but instead less than half the target value of

bytes are written to memory. Therefore another measurement was taken doubling the target value internally to achieve the correct counter values. The SIMD implementation does largely overcount and is therefore not usable combined with uncachable memory. C and ASM behave similar and come closer to the target value. The deviation though is still greater than with virtual process or shared memory.

		Bytes read		Deviation	
Impl.	Target	without factor	with factor	without factor	with factor
SIMD	$64 \cdot 10^6$	$30.56 \cdot 10^6$	$91.01 \cdot 10^6$	−52.25 %	42.40 %
ASM	$64 \cdot 10^6$	$30.59 \cdot 10^6$	$61.16 \cdot 10^6$	−52.20 %	−4.44 %
C	$64 \cdot 10^6$	$30.58 \cdot 10^6$	$61.16 \cdot 10^6$	−52.22 %	−4.44 %

Table 7.12.: *Bytes written results for single process with uncachable memory*

Multithreaded

The results for virtual process memory are shown in Figure 7.7. The implementations do not work well in a multi process environment. Only the SIMD implementation without randomness and four processes coming close to the target value for a memory size of 512 MiB. Yet for other memory sizes it is the least accurate implementation. This suggests that it might be an outlier. Using shared memory brings no improvement as the results in Figure C.18 show. Only the fluctuations using a memory size of 16 MiB are less pronounced.

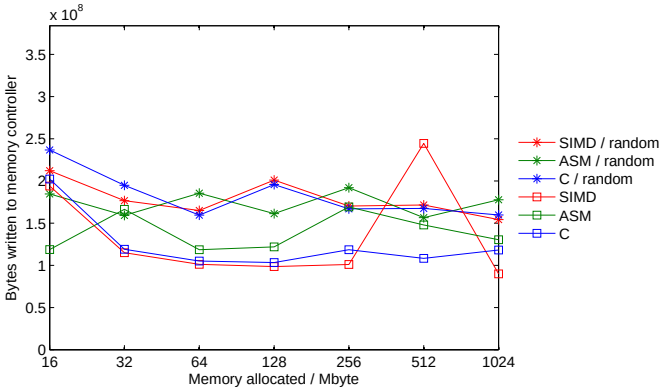
The kernel module performs worse if multiple processes are executed in parallel as shown in Table 7.13 and 7.14. While the C implementation seems to improve slightly if no correction factor is used, it is still too far off from the target value to be usable. The ASM implementation does only deteriorate by a small margin if four processes are used while the SIMD implementation seems to be the least viable option across all process counts. If the correction factor is applied on multiple processes, all implementations overcount by a large margin. This renders the kernel module unusable for the bytes written performance counter.

		Bytes read		Deviation	
Impl.	Target	without factor	with factor	without factor	with factor
SIMD	$256 \cdot 10^6$	$94.28 \cdot 10^6$	$964.36 \cdot 10^6$	−63.17 %	376.70 %
ASM	$256 \cdot 10^6$	$118.00 \cdot 10^6$	$766.23 \cdot 10^6$	−53.91 %	299.31 %
C	$256 \cdot 10^6$	$134.20 \cdot 10^6$	$831.98 \cdot 10^6$	−47.58 %	324.99 %

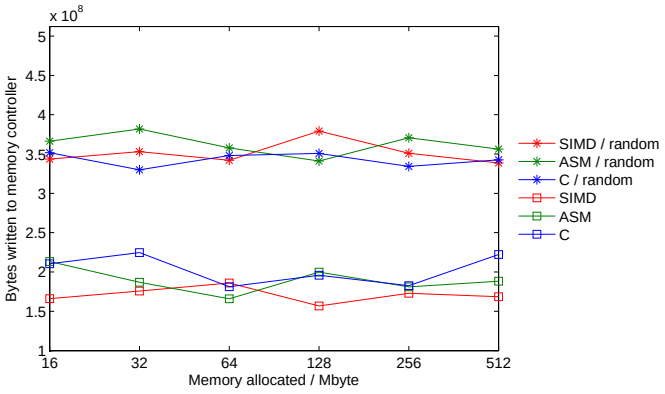
Table 7.13.: *Bytes written results for four processes with uncachable memory*

		Bytes read		Deviation	
Impl.	Target	without factor	with factor	without factor	with factor
SIMD	$512 \cdot 10^6$	$135.65 \cdot 10^6$	$2.79 \cdot 10^9$	−73.51 %	544.92 %
ASM	$512 \cdot 10^6$	$140.74 \cdot 10^6$	$2.46 \cdot 10^9$	−72.51 %	480.47 %
C	$512 \cdot 10^6$	$195.28 \cdot 10^6$	$2.44 \cdot 10^9$	−61.86 %	476.56 %

Table 7.14.: *Bytes written results for eight processes with uncachable memory*



(a) 4 processes



(b) 8 processes

Figure 7.7.: *Bytes written results for four and eight processes with virtual process memory*

As the results show, triggering the bytes written performance counter is struggling to achieve a counter value approximate to the target in multithreaded environments. Still the ASM implementation without randomness is selected as the most viable solution but care must be taken when this event trigger is used.

Side Effects

The side effects are presented in Table 7.15. All counters do exceed the background noise of the SUT. The L3 cache misses and L2 cache hits do exhibit fluctuations over different process counts. The large values for the read counter are not expected and show a decrease with rising process count. Using the bytes written counter in conjunction with bytes read would therefore need large corrections on the read counter. Interrupts increase if multiple processes are used but seem otherwise not affected by increasing the process count above the physical number of CPUs. The context switch counter on the other hand increases consistently with process count. Together with the negative outcome for multiple processes and large effects on the bytes read, this event trigger should be used with care.

	Processes		
Performance counter	1	4	8
L3 cache misses	$0.43 \cdot 10^{-4}$	$12.37 \cdot 10^{-4}$	$8.06 \cdot 10^{-4}$
L3 cache hits	$1.18 \cdot 10^{-3}$	$3.21 \cdot 10^{-3}$	$4.03 \cdot 10^{-3}$
L2 cache hits	$1.66 \cdot 10^{-3}$	$14.46 \cdot 10^{-3}$	$11.94 \cdot 10^{-3}$
Bytes read from memory controller	192.11	122.25	65.09
Instructions retired	129.09	129.93	130.58
Interrupts	$1.3 \cdot 10^{-5}$	$2.04 \cdot 10^{-5}$	$1.91 \cdot 10^{-5}$
Context switches	$1.71 \cdot 10^{-4}$	$9.34 \cdot 10^{-4}$	$18.58 \cdot 10^{-4}$
Average runtime (ms)	22	75.25	66.25

Table 7.15.: *Side effects of bytes written to memory controller per event generated*

7.1.6. Instructions Retired

The results of the instruction count trigger can be seen in Table 7.16. For each of the 10 000 transactions per process, $1 \cdot 10^6$ instructions had to be retired. The implementation for retired instructions works well with reasonable deviations below 1 %. An increase in overcount for multiple processes is expected and can be observed in the measurement results. As the overcount accumulates faster than the number of cores, a 10 times increase for an 8 times higher target value, it can be reasoned that this should be corrected for very high numbers of CPUs in a SUT to further improve the event trigger.

Processes	Target	Result	Deviation
1	$10 \cdot 10^9$	$10.005 \cdot 10^9$	0.05 %
4	$40 \cdot 10^9$	$40.121 \cdot 10^9$	0.30 %
8	$80 \cdot 10^9$	$80.414 \cdot 10^9$	0.52 %

Table 7.16.: *Retired instructions measurement results*

As shown, triggering the retired instruction counter works well on its own. Yet modeling a workload will make use of other performance counters as well. This will inevitably introduce an overcount of instructions retired if it is not corrected for.

Side Effects

The side effects are presented in Table 7.17. For a single process, all cache hit / miss counter stay below the system background noise. Yet the amount of side effects increases for these counters when multiple processes are executing. If four processes are executed, the L3 cache hits climb above the background noise. In case of eight processes, only the L2 cache misses stay below. Bytes read and written, together with the system wide counters context switches and interrupts are above the baseline for all number of processes executed.

	Processes		
Performance counter	1	4	8
L3 cache misses	$0.14 \cdot 10^{-7}$	$0.49 \cdot 10^{-7}$	$1.03 \cdot 10^{-7}$
L3 cache hits	$2.05 \cdot 10^{-7}$	$5.29 \cdot 10^{-7}$	$6.20 \cdot 10^{-7}$
L2 cache hits	$5.05 \cdot 10^{-7}$	$5.47 \cdot 10^{-7}$	$11.18 \cdot 10^{-7}$
Bytes read from memory controller	$1.67 \cdot 10^{-5}$	$11.97 \cdot 10^{-5}$	$20.09 \cdot 10^{-5}$
Bytes written to memory controller	$0.99 \cdot 10^{-5}$	$31.26 \cdot 10^{-5}$	$55.71 \cdot 10^{-5}$
Interrupts	$2.96 \cdot 10^{-8}$	$4.90 \cdot 10^{-8}$	$5.24 \cdot 10^{-8}$
Context switches	$7.75 \cdot 10^{-8}$	$9.13 \cdot 10^{-8}$	$11.63 \cdot 10^{-8}$
Average runtime (ms)	2811	2953.25	3992.63

Table 7.17.: *Side effects of instructions retired counter per event triggered*

7.1.7. Context Switches

Context switches can only be measured on a system wide basis. If several workloads are running in parallel, the average of the context switches for each measurement is taken because it could not be guaranteed that all workloads finish at the exact same time and all processes reading the counter are accessing the same value. To test if the factor of two is correct, a second measurement without the correction factor is executed for comparison.

The results for 10 000 transactions with 10 context switches each is shown in Table 7.18. It exhibits that the correction factor of two, described in Section 5.4, was wrongfully assumed with a deviation of nearly −30 % for single process and even higher for four and eight processes. A deviation of around 20 % could be measured without the correction factor. Therefore the factor is removed from the implementation for context switches.

		Result		Deviation	
Processes	Context switches	with factor	without factor	with factor	without factor
1	100 000	70 350.0	120 641.0	−29.7 %	20.6 %
4	400 000	271 682.5	481 323.8	−32.1 %	20.3 %
8	800 000	470 264.6	940 653.3	−41.2 %	17.6 %

Table 7.18.: *Context switches measurement results*

As the results without doubling the event trigger count are about 20 %, it is tested if a factor of 0.8 can correct for overcounting of context switches. A third measurement is taken and presented in Table 7.19. The results show that it works well with one and four processes. With eight processes, a slightly higher deviation can be observed but is still performing better as before. Therefore the correction factor of 0.8 is used in the framework to achieve a higher accuracy when generating context switches.

Processes	Context switches	Result	Deviation
1	100 000	100 588.0	0.6 %
4	400 000	400 688.5	0.2 %
8	800 000	757 056.0	−5.4 %

Table 7.19.: *Context switches measurement results with correction for overcounting*

Side Effects

Triggering a context switch is bound to have major side effects on other performance counters. Table 7.20 shows other implemented counter values as context switches are performed. L3 and L2 cache hits stay relatively close for one and four processes and only increase when eight processes are used. The L3 cache misses on the other hand, already have an increase when stepping from one to four processes but with relatively low values compared to cache hits. Despite expecting constant values for all process counts, bytes read increases significantly when multiple processes are executed. The bytes written counter exhibits a similar but less pronounced behavior. The memory access could be due to storing and loading the context information from main memory. The interrupt counter shows a fluctuating behavior for different process counts. All side effects are above the background noise in all three cases.

	Processes		
Performance counter	1	4	8
L3 cache misses	6.67	9.88	10.41
L3 cache hits	45.89	48.90	73.68
L2 cache hits	184.60	164.45	262.78
Bytes read from memory controller	5262.89	7177.48	7604.78
Bytes written to memory controller	4270.92	4624.21	4732.31
Instructions retired	23 742.65	23 572.34	23 620.37
Interrupts	$4.46 \cdot 10^{-3}$	$2.75 \cdot 10^{-3}$	$10.52 \cdot 10^{-3}$
Average runtime (ms)	559	705.50	1004.63

Table 7.20.: *Side effects of context switches per event generated*

7.1.8. Interrupts

The evaluation of interrupts triggered uses 10 interrupts per transaction which should be measured. The results in Table 7.21 show that the interrupt implementation performs well with only minor deviations from the target value. The minor decline in achieved interrupts for increasing number of processes should be further investigated in the future to be able to improve the implementation. As triggering interrupts has one of the longest runtimes, the decrease could be related to the relatively long time to program the APIC.

Side Effects

Table 7.22 shows the side effects of triggering hardware interrupts. Similar to context switches (see Section 7.1.7), interrupts also introduce major side effects. This is due to the fact that interrupts do cause two context switches per event. All side effects exceed the background noise. The cache hits and misses are increasing with a rising process count. This is contrary to the expectation of staying constant within reasonable deviations.

Processes	Interrupts	Result	Deviation
1	100 000	100 276.0	0.3 %
4	400 000	400 946.0	0.2 %
8	800 000	798 682.3	−0.2 %

Table 7.21.: *Interrupt measurement results*

Compared to a single process, the bytes read counter increases by over 50 % if four processes are used. The write counter on the other hand, does not increase if four processes are used. Yet this changes drastically when eight processes are executing and a major increase in the bytes read and written can be observed. This implementation should be used with care if more processes produce interrupts than the CPU has physical cores. As the side effects on context switches shows a constant behavior, this implementation might also be used to trigger context switches in pairs with an interrupt as side effect.

	Processes		
Performance counter	1	4	8
L3 cache misses	0.064	0.142	0.341
L3 cache hits	0.224	0.410	4.637
L2 cache hits	8.646	11.219	41.690
Bytes read from memory controller	50.64	79.82	265.37
Bytes written to memory controller	26.45	23.61	119.38
Instructions retired	15 258.97	15 320.79	15 385.13
Context switches	2.00	2.00	2.01
Average runtime (ms)	6487	6453.50	6528.38

Table 7.22.: *Side effects of interrupts per event triggered*

7.2. Performance Event Trigger Framework

This section describes the evaluation of the Performance Event Trigger Framework (PET) in terms of feasibility and accuracy. First the local reference workloads Pi, XMLValidate and SSJ are evaluated. Afterwards the NFV workload is evaluated. Reference measurements are taken with 10 load levels, ranging from 10 % to 100 %. For each load level, a pre-measurement, measurement and post-measurement time of 30 s, 120 s and 10 s is used. All measurements with PET approximating the workloads are measured for three composition mechanism, the naive method without side effects, the accumulation method and simulated annealing described in Section 4.2.2. The measurements include the power consumption, score and transaction count, recorded by Chauffeur and the performance counters for L3 cache misses and hits, read and written bytes, instructions retired, context switches and interrupts.

To further improve the implementation, measurements are taken, in which some performance counters are not triggered. A performance counter is removed under the assumption of a low or negative impact on the power consumption, if

- its measurement is overcounting by at least one order of magnitude, taking into account the median over all load levels.

- its correlation of the reference measurement with the power consumption is below a value of 0.9.
- its median reference value is below the background noise shown in Table A.1.

The measurement is considered invalid and not evaluated if a stable load level could not be reached. A load level is stable if the score, transaction count divided by the measurement phase's runtime, does not deviate largely from the target score necessary for the load level. All power measurement results include the 95 % confidence interval. To determine the best solution for the workload, the mean and maximum deviation and Coefficient of Variation (CV) are compared. If no clear solution can be determined, the accuracy of the performance counters are compared.

The side effects measured in Section 7.1 are rounded and transferred to the configuration of PET. The configured factors are shown in Table D.2. All side effects are incorporated to increase the accuracy because PET uses 8 processes for evaluation and all side effects for the event triggers exceed the background noise.

As expected, PET achieves a good accuracy on focused workloads like Pi. The accuracy decreases with the rising complexity of the workload but still has a reasonable accuracy for the NFV workload. All workloads can be approximated with a mean deviation of less than 10 %. Which of the three composition mechanisms should be selected and if performance counters should be removed is dependent on the workload. The decision must therefore be left to the user to obtain an accurate approximation.

7.2.1. Pi Workload

The Pi workload is CPU heavy and does not stress other hardware components. The results of the power measurements are shown in Figure 7.8 and Table 7.23. The measurements using no side effects and accumulation seem accurate on average, but the maximum deviation is over 10 %. The CV on the accumulation method is higher but it deviates less from the target power consumption on average and at its peak. It also has an invalid load level at 10 %. As neither of the both measurements is clearly superior, a decision must be reached considering the reduced configuration. Best results could be achieved at the medium load levels as PET tends to underestimate the power consumption on higher load levels and overestimation on lower load levels. Simulated annealing did not work as expected and misses the targeted power consumption by a large margin. It is monotonically increasing over the load levels but at a slow rate, resulting in bad approximations.

	Deviations from target		
Measurement	Mean	Maximum	CV
Naive	102.95 %	114.96 %	6.94 %
Accumulation	98.07 %	87.57 %	7.24 %
Simulated Annealing	57.95 %	40.37 %	28.26 %

Table 7.23.: *Pi workload mean and maximum deviation from the target and CV*

To improve accuracy, it is considered removing performance counters from the configuration fitting one of the three rules mentioned earlier. The correlations of the reference performance counter values with power consumption are listed in Table 7.24. Also included are the factor of the reference performance counters against the background noise and the factor of PET measurements against the reference.

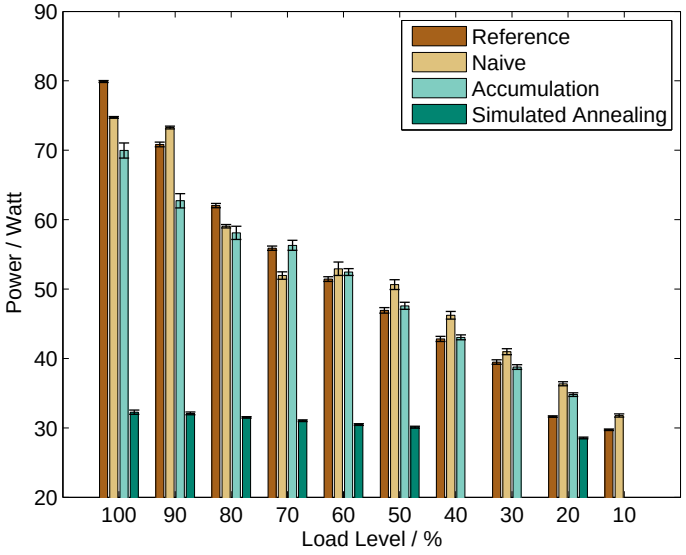


Figure 7.8.: *Pi workload power consumption*

As can be seen, all performance counters, except instructions retired and context switches can be removed from the configuration for all three implementations, to evaluate if the overall accuracy of the framework can be improved.

Performance counter	Correlation	Background noise factor	Target value factor Without SE	Accu.	Sim.An.
L3 cache misses	0.9087	0.0310	454.5196	404.4826	518.5293
L3 cache hits	0.9096	0.0029	621.5461	532.2341	673.4158
Bytes read	0.9789	1.0789	348.9333	142.3288	175.9248
Bytes written	0.9870	2.2438	250.5546	166.9880	159.7027
Instructions retired	0.9886	3.4061	1.0826	1.0766	0.1542
Interrupts	0.2186	2.3188	0.6848	0.5958	6.3017
Context switches	0.9726	24.3096	0.6783	0.6337	1.3057

Table 7.24.: *Pi workload performance counter results*

The results of the reduced configuration are illustrated in Figure 7.9 and Table 7.25. They show an overall improvement for all three measurements. The mean deviation from the target with the naive approach is only better by a small margin while its CV shows a noticeable decrease, together with the maximum deviation. Using side effects on the other hand improves on mean and maximum deviation as well as CV. Simulated annealing is considerably better using the reduced configuration with values in the same range as the naive and accumulation measurements. As before, the accumulation and naive solution are still the better options in a simple workload. The reduced solution with accumulated side effects is determined as the best solution, as its average deviation is closest to the target value with only marginally higher maximum deviation and CV.

It is shown that local workloads focusing on the CPU can be approximated well with good accuracy and only minor deviations from the target value below 5 %. If simulated annealing is used, the deviation is still below 10 % and therefore within a reasonable limit.

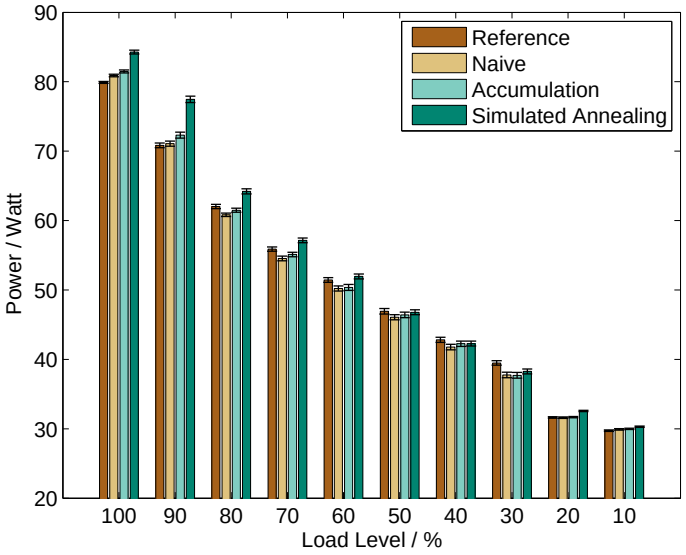


Figure 7.9.: *Pi workload power consumption with reduced configuration*

	Deviations from target		
Measurement	Mean	Maximum	CV
Naive	98.68 %	95.61 %	1.79 %
Accumulation	99.38 %	95.49 %	2.00 %
Simulated Annealing	102.18 %	109.36 %	3.45 %

Table 7.25.: *Pi workload mean and maximum deviation from the target and CV with reduced configuration*

7.2.2. XMLValidate

The SERT workload XMLValidate not only stresses the CPU but also the memory. It is therefore a good expansion on Chauffeur's Pi workload to evaluate if the accuracy demonstrated in Section 7.2.1 can be achieved on a more complex workload. The power measurements with all performance counters is shown in Figure 7.10. Most load levels of the naive measurement are marked as invalid. Simulated annealing also has five invalid load levels. With only three valid load levels on the naive measurement, caution must be taken not to over emphasize on the sparse data set available.

The lower load levels of the valid data reach the target value with minor deviations. At the 100 % load level, the naive method is at its maximum deviation and it is therefore unlikely that it could reach other load levels with reasonable accuracy. Accumulation works well and reaches deviations and a CV comparable to the Pi workload. Simulated annealing has the same problem already encountered in the Pi workload of not reaching the target power consumption by a large margin.

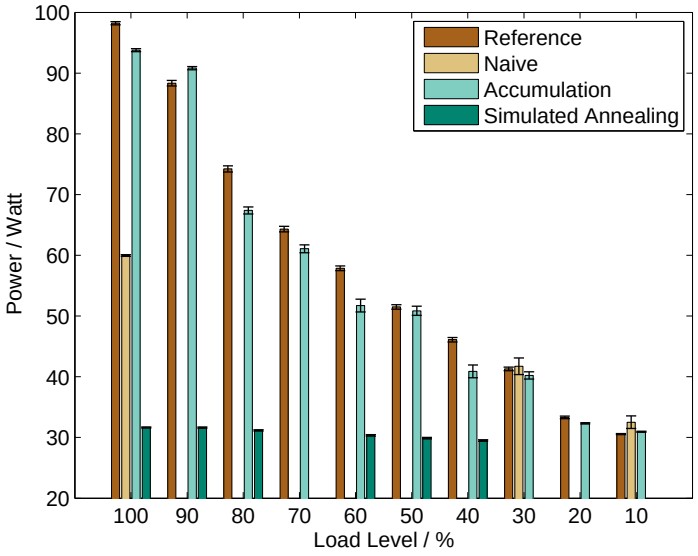


Figure 7.10.: *XMLValidate workload power consumption*

	Deviations from target		
Measurement	Mean	Maximum	CV
Naive	89.52 %	61.07 %	27.69 %
Accumulation	95.68 %	88.71 %	5.04 %
Simulated Annealing	47.43 %	32.22 %	26.81 %

Table 7.26.: *XMLValidate mean and maximum deviation from the target and CV*

As with the Pi workload, performance counters are removed from the configuration to see if the first results can be improved, based on Table 7.27. The L3 cache misses and hits, bytes read and written and the instruction count are kept. They all have a high correlation, are above the background noise and none of them is overcounting by a large margin. The interrupts are removed from all configurations as it has a low correlation, yet higher compared to the Pi workload. The context switches are used for the simulated annealing

and accumulation measurements. They are removed from measurements including side effects due to a large overcount.

Performance counter	Correlation	Background noise factor	Target value factor Without SE	With SE	Sim.An.
L3 cache misses	0.9860	12.4535	0.0377	0.0145	0.8449
L3 cache hits	0.9852	1.2912	0.0622	0.0135	0.9668
Bytes read	0.9805	463.9503	1.8800	0.0488	0.3203
Bytes written	0.9829	865.1718	0.6841	0.0647	0.3417
Instructions retired	0.9829	5.8681	0.0694	0.6656	0.0840
Interrupts	0.4662	1.0403	0.4540	6.8549	16.0928
Context switches	0.9585	5.9283	0.0533	114.5332	5.6804

Table 7.27.: *XMLValidate performance counter results*

Reducing the performance counters does not yield the expected improvements. The naive measurement has only two valid load levels. A possible reason is the low score of 8.896 and a high client CV across all load levels (see Table E.3, E.4 and E.5). Using the accumulation approach in the reduced measurement did not improve the approximation. In fact, it stayed close to the measurement including all performance counters. As with the Pi workload, simulated annealing works better with less performance event triggers to optimize. Yet it is still off by over a quarter of the target power consumption.

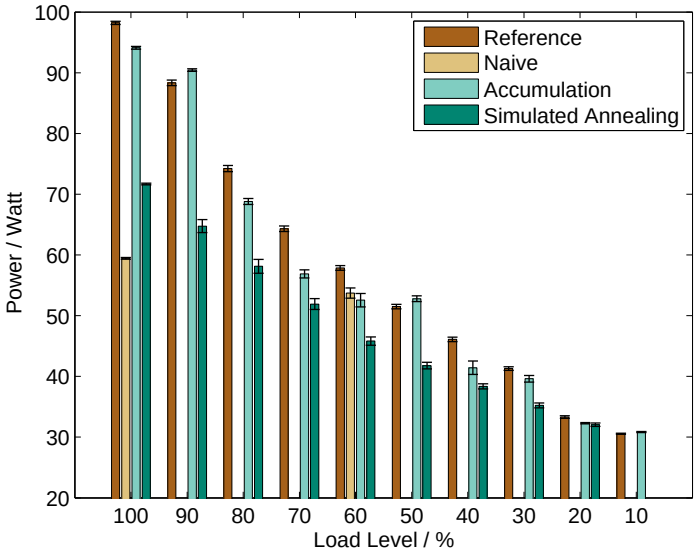


Figure 7.11.: *XMLValidate power consumption with reduced configuration*

Using a more complex workload does not necessarily result in a higher deviation. The accumulation measurement is in close proximity to the full configuration measurement taken for the Pi workload with a mean deviation of below 5 %. Yet reducing the amount of performance counters did not yield better results. This is mainly due to the workload stressing more hardware components which in turn leads to more performance triggers necessary to approximate the power consumption. While high deviations can be observed in the naive and simulated annealing measurement, it is reasoned that the behavior is due to the low score and therefore transaction count. This could prohibit PET reaching a stable throughput, which is necessary for the load level to be valid.

Measurement	Deviations from target Mean	Maximum	CV
Naive	76.69 %	60.52 %	29.82 %
Accumulation	95.64 %	88.43 %	5.40 %
Simulated Annealing	81.15 %	72.98 %	8.61 %

Table 7.28.: *XMLValidate mean and maximum deviation from the target and CV with reduced configuration*

7.2.3. SSJ

The SSJ is the most complex local workload, executing different transactions. The results for SSJ are shown in Figure 7.12 and Table 7.29. Neglecting the side effects does not yield good results. As with XMLValidate, most load levels are invalidated due to large deviations in the score. The naive approach is therefore not suitable. To shorten the runtime of a transaction, the configured event count could be reduced. Yet this would result in some event trigger counts falling below one. As the current implementation of PET only supports integer values, this is not applicable. Incorporating the side effects in the measurement results in all load levels being valid. Yet it always exceeds the reference power consumption. In this complex workload, most performance counter values are configured to zero as the side effects when the accumulation method is used often outweigh the event trigger count and no performance events are triggered at all. This will cause PET to end a transaction quickly, stressing the CPU, the dominant power consumer, and subsequently results in the observed overshot in power consumption. It also causes the counter values to be significantly below one in four of the seven counter values, shown in Table 7.30. Simulated annealing balances the side effects and configured parameters and can achieve a reasonable approximation that is slightly above 5 % on average with the maximum deviation staying below 10 %.

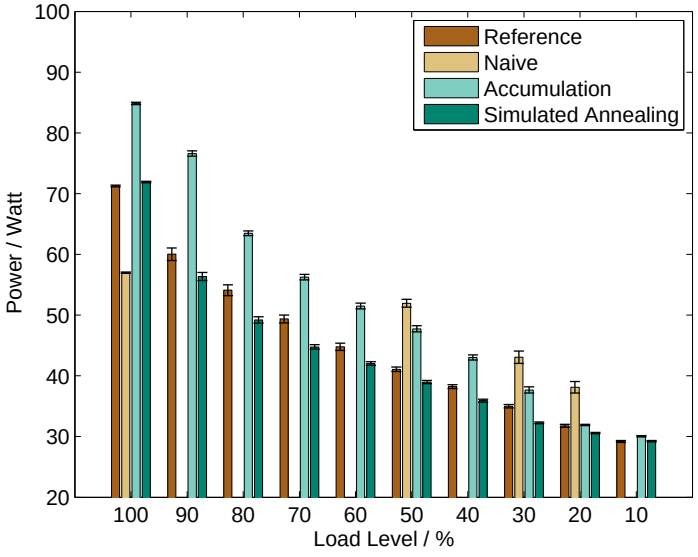


Figure 7.12.: *SSJ workload power consumption*

Measurement	Deviations from target Mean	Maximum	CV
Naive	112.35 %	126.44 %	19.37 %
Accumulation	113.28 %	127.61 %	7.03 %
Simulated Annealing	94.75 %	90.65 %	3.66 %

Table 7.29.: *SSJ mean and maximum deviation from the target and CV*

For the reduced configuration, only the interrupts and context switches are a viable option to remove, as they are below the background noise. All other performance counters have a high correlation, are above the background noise and do not overcount by one order of magnitude.

Performance counter	Correlation	Background noise factor	Target value factor Without SE	With SE	Sim.An.
L3 cache misses	0.9835	63.0325	0.0283	0.0905	0.8914
L3 cache hits	0.9752	6.5001	0.1186	0.1165	1.1638
Bytes read	0.9987	706.8520	1.0713	0.1638	0.9267
Bytes written	0.9841	531.5548	0.4502	0.0075	2.1231
Instructions retired	0.9789	1.6516	0.1856	1.5625	0.7991
Interrupts	0.9725	0.4753	1.3286	6.2710	1.6641
Context switches	0.9932	0.5965	0.2914	65.1277	54.9626

Table 7.30.: *SSJ performance counter results*

The results for the reduced configuration is presented in Figure 7.13 and Table 7.31 and show an improvement for the naive approach. Yet above the 50 % load level, an accurate and stable load level could not be achieved. Accumulating side effects did improve on average but not on CV and maximum deviation. The measurement still seems to suffer from the same problem observed in the full configuration measurement and is largely above the target power consumption at full load. The simulated annealing measurement also did not improve. Its accuracy decreases in both, mean and maximum deviation, and the CV increases.

Measurement	Deviations from target Mean	Maximum	CV
Naive	96.65 %	120.91 %	12.99 %
Accumulation	102.97 %	127.84 %	9.92 %
Simulated Annealing	90.37 %	82.63 %	7.38 %

Table 7.31.: *SSJ mean and maximum deviation from the target and CV with reduced configuration*

The results show that even a complex workload can be approximated with a reasonable accuracy below 10 % on both mean and maximum deviations. A mean deviation of below 5 %, as reached with XMLValidate and Pi could not be achieved. It also is determined that reducing the amount of performance counters to trigger can have a detrimental effect on the accuracy of the approximation when simulating complex workloads.

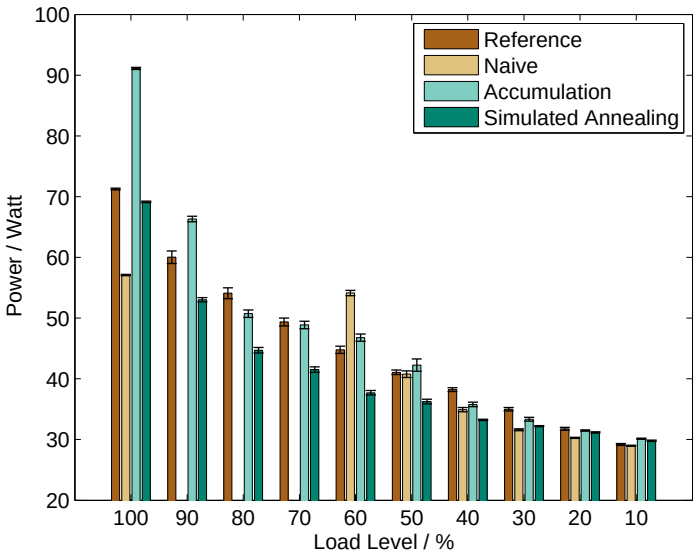


Figure 7.13.: *SSJ power consumption with reduced configuration*

7.2.4. NFV Workload

The NFV workload, a DPI firewall, stresses hardware parts otherwise not used by the local workloads evaluated before. In this section, the ability to approximate the power consumption without putting these specific hardware components under load is evaluated. For this workload, two NICs together with CPU and memory need to be approximated without stressing the NICs.

From the results shown in Figure 7.14 and Table 7.32, it can be determined that the naive measurement works best, deviating less in both mean and maximum from the target. Yet it is underestimating the power consumption consistently with one exception at the 90 % load level. The local workloads on the other hand, alter more often between over- and underestimating the power consumption. This behavior is expected and most likely stems from the NICs that cannot be stressed via the approximation. The accumulation and simulated annealing measurements could not achieve a good accuracy with high deviations and CVs.

	Deviations from target		
Measurement	Mean	Maximum	CV
Naive	93.37 %	83.65 %	8.27 %
Accumulation	76.31 %	59.81 %	14.33 %
Simulated Annealing	79.23 %	64.03 %	12.33 %

Table 7.32.: *NFV mean and maximum deviation from the target and CV*

According to Table 7.33, the reduced configuration is identical for all implementations. Only the L3 cache misses can be removed as they are below the background noise. Other performance counters either have a high correlation, are above the background noise and do not overcount by one order of magnitude.

Reducing the amount of performance events to trigger did not result in an overall better approximation as shown in Figure 7.15 and Table 7.34.

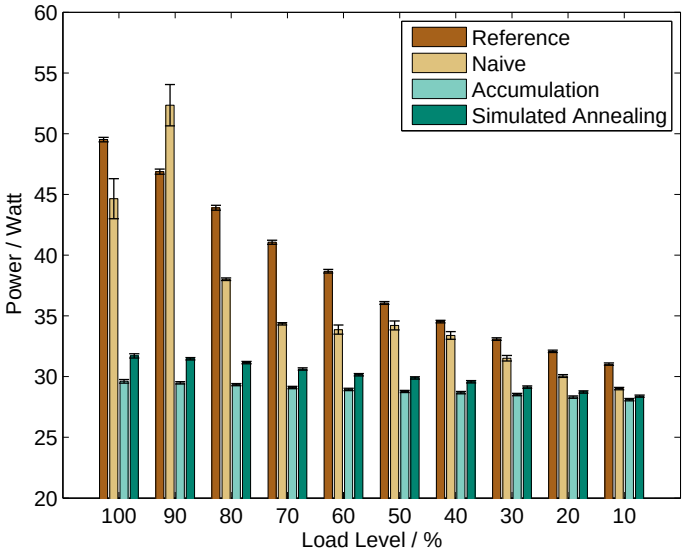


Figure 7.14.: *NFV workload power consumption*

Performance counter	Correlation	Background noise factor	Target value factor Without SE	With SE	Sim.An.
L3 cache misses	0.9677	2.7756	1.3799	0.0186	1.5721
L3 cache hits	0.9676	0.2600	1.7484	0.0210	1.7288
Bytes read	0.9429	72.2970	9.2149	0.0089	0.9960
Bytes written	0.9695	91.5291	5.0920	0.0102	1.3954
Instructions retired	0.9762	1.9273	0.2673	0.1463	0.2570
Interrupts	0.9967	44.3945	0.0696	0.5055	0.3108
Context switches	0.9917	103.7100	0.0822	0.3643	0.3045

Table 7.33.: *NFV workload performance counter results*

This coincides with the results from the XMLValidate and SSJ workloads. The accumulation and simulated annealing measurements only show minor changes and are still off by a large margin. The naive measurement only improved on average but at the cost of a higher CV and maximum deviation. The figure also shows the same overestimation at the 90 % load level already observed with the full configuration, which could be the result of a systematic problem or a low score and large variation among the clients executing the workload.

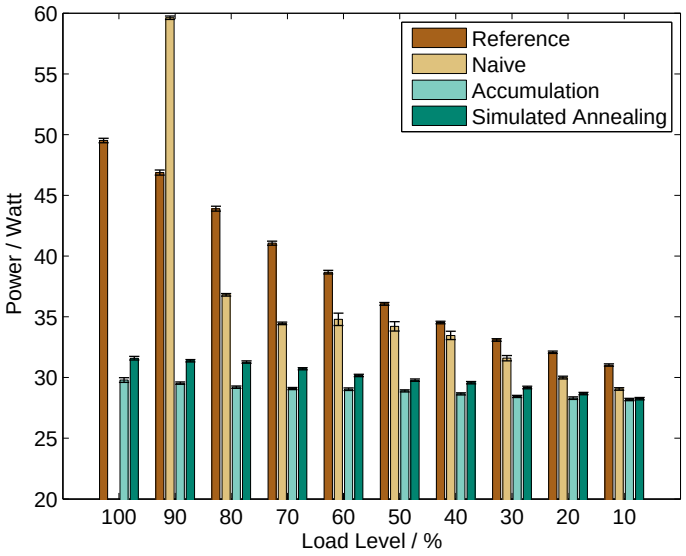


Figure 7.15.: *NFV power consumption with reduced configuration*

	Deviations from target		
Measurement	Mean	Maximum	CV
Naive	95.48 %	127.18 %	13.42 %
Accumulation	76.38 %	60.17 %	14.28 %
Simulated Annealing	79.17 %	63.80 %	12.28 %

Table 7.34.: *NFV mean and maximum deviation from the target and CV with reduced configuration*

The approximation measurements are repeated with an elongated measurement phase of 240 s. It is expected that a longer measurement phase could reduce the CV in the higher load levels as well as the score and therefore result in less invalid load levels. The results are shown in Figure 7.16 and Table 7.35.

While a longer measurement phase now allows for a more stable measurement at higher load levels, the mean deviation for the accumulation measurement has increased but is still below 10 %. Yet its CV and maximum deviation have significantly improved. The accumulation and simulated annealing measurement have not been influenced by the longer measurement phase. They suffer from the same problem already encountered at the shorter measurement phase.

It is expected after more stable load levels are reached that the naive measurement would converge towards the simulated annealing measurement shown in Figure 7.17. In this scenario, it would be clear from the increasing difference between the target target and

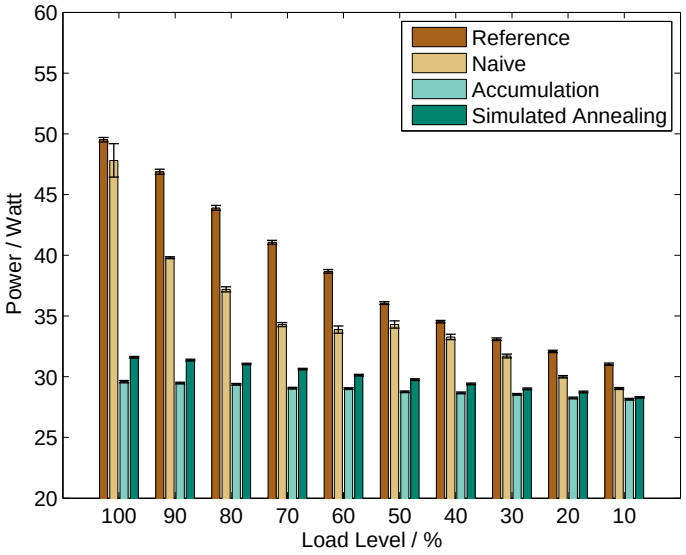


Figure 7.16.: *NFV power consumption with* 240 s *measurement phase*

	Deviations from target		
Measurement	Mean	Maximum	CV
Naive	91.16 %	83.53 %	5.86 %
Accumulation	76.32 %	59.77 %	14.33 %
Simulated Annealing	79.00 %	63.81 %	12.32 %

Table 7.35.: *NFV mean and maximum deviation from the target and CV with* 240 s *measurement phase*

the measured power consumption, that the NICs side effects on the power consumption are not approximated. In the current scenario, the naive measurement can approximate an externally driven workload with a reasonable accuracy of less than 10 % on average. Simulated annealing and accumulation are not optimal with larger deviations.

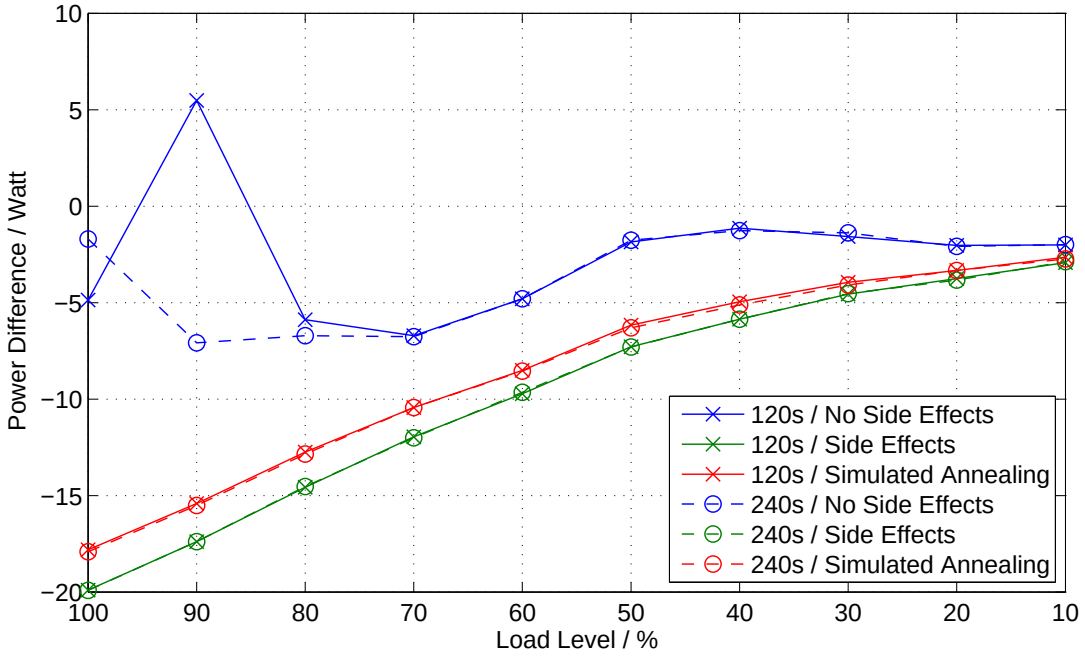


Figure 7.17.: *NFV workload power consumption differences between reference and approximation measurements*

Table 7.36 illustrates all evaluated workloads with their optimal configuration. For the NFV workload, the elongated measurement is selected despite a lower mean deviation which is caused by a large single outlier. All workloads can be approximated with an accuracy of 10 % with the simple Pi workload offering the best approximation, decreasing with complexity as expected.

			Deviations from target		
Workload	Configuration	Measurement	Mean	Maximum	CV
Pi	Reduced	Naive	99.38 %	95.49 %	2.00 %
XMLValidate	Full	Accumulation	95.68 %	88.71 %	5.04 %
SSJ	Full	Simulated Ann.	94.75 %	90.65 %	3.66 %
NFV	Elongated, Full	Naive	91.16 %	83.53 %	5.86 %

Table 7.36.: *Comparison of workloads with mean and maximum deviation and CV*

It has been shown that a workload stressing hardware components otherwise not used by locally executed workloads can be approximated while maintaining a reasonable accuracy. Performance counters are a feasible option to approximate workloads and no knowledge of the externally driven component is necessary. Yet care must be taken which implementation should be used and if the workload is specialized enough, like the Pi workload, to justify removing performance counters from the configuration without detrimental effects.

7.2.5. Lower Bound for Valid Measurements

Some measurements are marked as invalid and could not be included in the evaluation. Those invalid measurements occur mostly in runs not including side effects and therefore less transactions can be executed in the same time span due to longer runtimes. To determine the cause for this behavior, the calibrated scores, number of transactions executed divided by the measurement time for the full and reduced configuration measurements are summarized in Table 7.37. It shows that invalid measurements are more likely to happen if the score is low. It is assumed that there is a bound to the transaction count, below PET might not achieve a stable load level and subsequently a valid measurement. CVs between the client processes on the SUT are higher and the scores are lower than their valid counterparts, which can be seen in Table E.3, E.4 and E.5, supporting the assumption of a lower bound. A lower bound on the transaction count would also imply an upper bound on the transactions size. Transaction size is determined by the amount of performance events to trigger. More events to trigger causes the transaction to have a longer runtime resulting in a lower transaction count or score for a given measurement time.

Workload	Configuration	Measurement	Score	Valid Load Levels
Pi	Full	No Side Effects	381.809	10
		Side Effects	488.154	9
		Sim. An.	52.707	7
	Reduced	No Side Effects	35 980.271	10
		Side Effects	40 606.948	10
		Sim. An.	21 134.736	10
XMLValidate	Full	No Side Effects	8.896	3
		Side Effects	1 157 616.680	10
		Sim. An.	61.728	6
	Reduced	No Side Effects	8.896	2
		Side Effects	1 165 138.268	10
		Sim. An.	154.047	9
SSJ	Full	No Side Effects	18.101	4
		Side Effects	61 481.960	10
		Sim. An.	1408.967	10
	Reduced	No Side Effects	1809.804	7
		Side Effects	1 467 523.797	10
		Sim. An.	139 141.733	10
NFV	Full	No Side Effects	587.016	10
		Side Effects	6261.657	10
		Sim. An.	2331.591	10
	Reduced	No Side Effects	574.314	9
		Side Effects	6213.617	10
		Sim. An.	2360.121	10

Table 7.37.: *Calibrated workload scores compared to valid load levels*

To determine the lower bound, the scores of all invalid measurements from all workloads are grouped in a histogram. Outliers with values above a score of 300 are removed and the result is shown in Figure 7.18. It can be seen that a lower score results in a higher number of invalid measurements and a lower bound exists at a score of 25.

Despite this relatively low bound, transaction count should be significantly higher to completely avoid invalid measurements, as Table 7.37 shows.

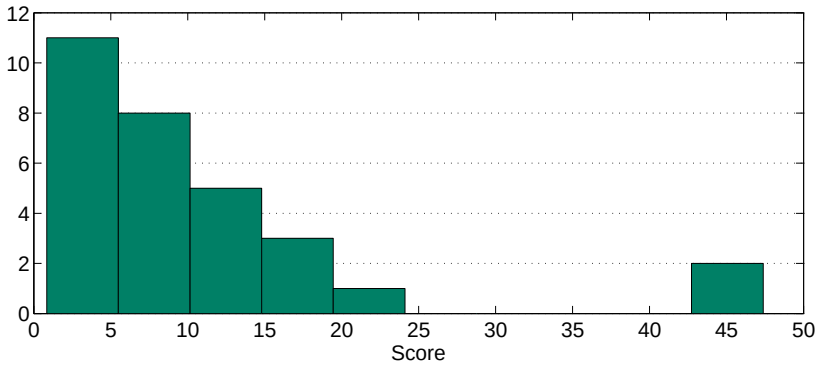


Figure 7.18.: *Histogram of invalid measurements by score*

Avoiding the lower bound can be achieved in two ways. A transaction could be shortened by reducing the amount of performance events to trigger, which would reduce the runtime and increase the transaction count. Yet this might not be feasible if a configuration parameter falls below one, because PET can only handle integer values. The second more promising option, used in the NFV workload, is to increase the measurement time.

7.3. Linear Regression Model

In this section, the linear regression model described in Section 4.3 is evaluated if it can predict the power consumption of the NFV workload. The model is fitted to the data from the reference measurements of the Pi, XMLValidate and SSJ workloads. The four measurements achieving the best approximation for each workload, listed in Section 7.2.4, are used for the prediction. The results are shown in Table 7.38.

	Workload			
Load Level	Pi	XMLValidate	SSJ	NFV
100 %	−92.26	−755.22	−923.29	−493.85
90 %	−80.70	−657.58	−746.59	−453.25
80 %	−68.34	−581.71	−642.79	−405.90
70 %	−55.71	−510.63	−552.00	−358.60
60 %	−43.03	−438.95	−470.24	−305.35
50 %	−30.30	−364.76	−376.26	−250.23
40 %	−17.52	−291.76	−290.08	−196.34
30 %	−4.91	−220.05	−211.71	−140.32
20 %	4.93	−147.00	−122.42	−84.59
10 %	14.20	−76.37	−39.25	−29.96

Table 7.38.: *Power consumption prediction of the linear regression model in watts*

It is obvious that the model did not work correctly for all workloads and the prediction is even inverse to the load levels. To improve the model and remove detrimental effects, a t-test is performed to identify performance counter coefficients that have no significance in the model and remove them. The t value for each coefficient is calculated and tested

against the null hypothesis with a significance level of $\alpha = 0.05$. Only the bytes written to memory controller counter is not significant and removed. The obtained coefficients and standard errors of the modified model are shown in Table F.6 and the results are listed in Table F.7. The results show that no improvement could be reached and the model is still inverse.

As the workloads selected are increasing in complexity and the Pi workload in particular is only stressing the CPU, performance counter variation is high. Table 7.39 shows the CV of each performance counter for different combinations of workloads. It is assumed that a different combination of the workloads available, with lower CVs, might yield better prediction for the local workloads.

Table 7.39 contains the CV of the observations for different workload combinations. No combination has CV values deemed sufficiently low. Yet the combination of SSJ and XMLValidate seems the most promising base for the model.

	Workload Combination			
Performance Counter	Pi SSJ XMLValidate	Pi SSJ	Pi XMLValidate	SSJ XMLValidate
L3 cache misses	168.05	127.63	155.17	125.17
L3 cache hits	173.78	127.31	160.03	130.36
Bytes read	114.57	125.73	142.69	73.65
Bytes written	102.32	126.76	128.47	60.32
Instructions retired	75.84	61.55	66.75	83.87
Interrupts	65.09	45.42	70.86	44.74
Context switches	119.00	82.45	116.12	101.07

Table 7.39.: *Performance counter CV of different workload combinations in percent*

The coefficients for the model based on XMLValidate and SSJ and the prediction are shown in Table F.8 and 7.40. While the results for the Pi workload are more promising, they are still off by a large margin. The other workloads do not work either and show no promising results. The model also seems to exhibit a strange behavior for the local workloads in which it inverts its direction. This can be observed well on the SSJ workload around the 50 % load level, where the power consumption declines until the 50 % load level, reaching its minimum, and then increasing towards higher load levels.

As no promising model could be found, based on [AKK+09] and [BKW05], the model is analyzed for multicollinearity. It is present if performance counters in the model are not linearly independent. To test for multicollinearity, the condition index ξ and the influence of each performance counter on the variance of the estimator is evaluated. If more than 50 % of the variance can be asserted to two or more coefficients of a condition index ξ_j, a multicollinearity between the parameters can be assumed. It can also be present if ξ is between 5 to 10 for weak and between 30 to 100 for moderate to strong multicollinearity. The condition index is calculated by $\xi = \sqrt{\lambda_{max}/\lambda_j}$. λ_j are the eigenvalues of the matrix $\boldsymbol{X'X}$ (see Equation 4.6). The results for the multicollinearity analysis are shown in Table 7.41. As can be seen, the 8th eigenvalue accounts for more than 50 % in all performance counters indicating a multicollinearity between them. The 6th, 7th and 8th eigenvalue also show a high ξ which also indicates multicollinearity.

The proposed linear regression model did not work and could not be used for the evaluation described in Section 4.3. The proposed model's prediction is inverse to the load levels

		Workload		
Load Level	Pi	XMLValidate	SSJ	NFV
100 %	54.86	−740.72	21.42	−102.06
90 %	36.63	−701.46	4.29	−90.82
80 %	30.92	−636.72	0.60	−79.27
70 %	24.26	−587.72	−5.22	−67.78
60 %	18.14	−534.20	−10.47	−54.49
50 %	12.84	−521.91	−16.64	−41.03
40 %	10.80	−400.15	−14.83	−27.86
30 %	10.52	−368.75	−8.47	−14.19
20 %	11.55	−329.89	−5.13	−0.40
10 %	13.54	−342.84	1.47	14.26

Table 7.40.: *Power consumption prediction of the linear regression model in watts*

and does not predict the power consumption correctly, even after non-significant performance counters are removed. Evaluating the CV of different combinations of local workloads showed high variation among the observations. Despite selecting the most promising combination, the model could not predict the power consumption of any workload with reasonable accuracy. The analysis also indicated that there is a strong multicollinearity between the performance counters. This leads to the conclusion that a linear regression model is insufficient for predicting the power consumption through performance counters.

Dimension j	ξ	L3 misses	L3 hits	Bytes read	Bytes written	Instructions ret.	Interrupts	Context switches
1	1	0.000	0.000	0.000	0.000	0.000	0.000	0.000
2	1.57	0.000	0.000	0.000	0.000	0.000	0.000	0.000
3	2.86	0.000	0.000	0.000	0.000	0.000	0.001	0.000
4	7.76	0.000	0.000	0.000	0.000	0.000	0.025	0.001
5	14.7	0.000	0.000	0.000	0.000	0.000	0.010	0.000
6	35.7	0.000	0.000	0.006	0.001	0.002	0.001	0.004
7	252	0.002	0.000	0.191	0.491	0.113	0.067	0.121
8	1430	0.998	0.999	0.803	0.508	0.885	0.896	0.875

Table 7.41.: *Multicollinearity analysis*

8. Conclusion

Due to the rising amount of servers and network equipment, energy demand in today's IT infrastructure rises consistently. To handle this problem, efficiency measurements that aid in the selection of suitable appliances are necessary. SPEC offers the SERT benchmark to rate the energy efficiency of servers to identify the best suited equipment based on standardized workloads. Yet modern IT infrastructure is highly interconnected and the workload is usually not run in isolation on a single machine. With the rise of SDN and NFV these networks become more flexible and off-the-shelf servers are taking over tasks normally carried out by dedicated network appliances.

Efficiency in networks is often viewed on a broader scope, taking multiple machines and network functions into account. To achieve a lower power consumption while being able to satisfy the demanded load, the placement and distribution of network functions is researched, including consolidating and splitting up virtualized functions. Another point of view is the component level, rating specific hardware parts for energy efficiency. Models, often using performance counters, are used to predict the power consumption of a system. Yet an approximation of the power consumption as a workload is not available.

Existing benchmarks are either run locally or need complex testbeds to stress the SUT, which needs expertise and is time consuming. Locally executed benchmarks on the other hand do not use hardware components otherwise put under load by external requests. To produce reliable performance measurements with a local benchmark including hardware normally not used by a local workload, the Performance Event Trigger Framework (PET) is proposed. Its goal is to approximate the power consumption which would normally be observed on interconnected systems under different load levels while being compatible with Chauffeur, also used by the SERT. PET triggers performance counter events to simulate the workload.

As benchmarks must be repeatable and reliable, the first step is to identify which performance counters are suited by correlation analysis. From the available counters, L3 / L2 cache hits and misses, bytes read from and written to memory controller, instructions retired, context switches and interrupts are selected due to their high correlation with power consumption. Afterwards the implementations that should reliably trigger events to achieve accuracy is presented.

The implementations are evaluated on how accurate and therefore reliable they can approximate a performance counter. The optimal solution is selected and integrated into PET. During the evaluation of the selected performance counters, it became clear that

triggering L2 cache hits did not work with the proposed implementation and is therefore not part of PET. The evaluation showed that most counters can be implemented with reasonable accuracy. Yet especially memory related counters tend to be less accurate when used in a multithreaded environment. It is also shown that triggering performance counters comes at a cost. Each performance event does introduce side effects when triggered. These side effects are also evaluated.

To validate the implemented framework three local workloads, Pi from Chauffeur, XML-Validate and SSJ from the SERT are used. They range in complexity from simple CPU heavy calculations in the Pi workload, to a diverse hardware usage executing different operations in SSJ. As an exemplary workload that is dependent on external devices, a DPI firewall, acting as a VNF is set up. For each of the four workloads, three different approaches for the side effects are evaluated, including measurements ignoring them, accumulating the side effects and remove them from the amount of events to trigger and simulated annealing, balancing the side effects with the configured event count. Further measurements are taken to evaluate if performance counter can be removed that might have detrimental effects on the approximation.

Also a linear regression model is created to evaluate the approximation with the predictions. The model is build upon the observations from the three local workloads. Yet it turned out to be unfeasible to produce correct predictions even for the local workloads it is build upon. Removing insignificant parameters did not resolve the issue. Choosing a different combination with a lower CV among the observations could also not remedy the prediction problems. The performed analysis indicated multicollinearity between parameters and the model is deemed unfit for use with the selected performance counters.

The evaluation showed that workloads can be approximated with an average deviation of less than 10 %. On very focused workloads, such as Pi, removing performance counters has a positive impact on the approximation, reducing the average deviation to below 1 %. Yet the remaining more complex workloads did not profit as much due to their higher complexity. This shows that approximating locally and externally driven workloads, specifically NFV workloads, is possible with reasonable accuracy using the Performance Event Trigger Framework.

During the evaluation, it became apparent that there is a lower bound on the transaction count, below no stable load level can be reached. This bound is identified and available solutions are presented.

8.1. Future Work

This thesis can be used as a basis for future research of performance counters, power consumption approximation and modeling, and energy efficiency. In the course of this thesis, new questions arose of which a few will be listed here.

- Implement new performance event trigger to have a more diverse set. This will not only allow PET to have a better approximation but also can widen the possible workloads that can be approximated.
- Extend PET to not only allow linear correlation but also others, like exponential for example. Possible variants could also include a decrease in events triggered with increasing load levels.
- Research the side effects of the performance counters further to better understand how they occur and possible interactions between them.

- Code analysis in combination with measurements of workloads could be used to define a model that can represent the amount of performance events that would be generated for certain operations or code segments. PET would then be able to simulate a workload based on the source code.
- Find a solution for keeping the transaction runtime close to the approximated framework while maintaining a good approximation to make PET measurements compatible with already existing efficiency measurements of the same workload.
- PET can be used for faster resource planning and estimation.

Bibliography

[AKK+09] S. Albers, D. Klapper, U. Konradt, A. Walter, and J. Wolf, *Methodik der empirischen Forschung*. Gabler Verlag, 2009.

[AMD16] AMD, *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Advanced Micro Devices Inc., April 2016. [Online]. Available: http://support.amd.com/TechDocs/24593.pdf

[BB10] A. Beloglazov and R. Buyya, "Energy efficient resource management in virtualized cloud data centers," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010.

[BBLM14] R. Bolla, R. Bruschi, C. Lombardo, and S. Mangialardi, "Dropv2: Energy-efficiency through network function virtualization," *IEEE Network*, vol. 28, no. 2, pp. 26–32, Apr. 2014.

[BBN+] T. Bowden, B. Bauer, J. Nerin, S. Feng, and S. Seibold, "The /proc filesystem," accessed: 15.09.2016. [Online]. Available: http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/proc.txt

[Bel00] F. Bellosa, "The benefits of event-driven energy accounting in power-sensitive systems," in *EW 9 Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, 2000, pp. 37–42.

[BHD+12] J. F. Botero, X. Hesselbach, M. Duelli, D. Schlosser, A. Fischer, and H. de Meer, "Energy efficient virtual network embedding," *IEEE Communications Letters*, vol. 16, no. 5, pp. 756–759, Mar. 2012. [Online]. Available: http://www.sahandtarjomeh.com/wp-content/uploads/2015/09/Energy-Efficient-Virtual_d7rf4e8w5f41s3rf01f.pdf

[BJ12] W. L. Bircher and L. K. John, "Complete system power estimation using processor performance events," *IEEE Trans. Comput.*, vol. 61, no. 4, pp. 563–577, Apr. 2012. [Online]. Available: http://dx.doi.org/10.1109/TC.2011.47

[BKW05] D. Belsley, E. Kuh, and R. Welsch, *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity*, ser. Wiley Series in Probability and Statistics. Wiley, 2005.

[BLCC15] M. Bouet, J. Leguay, T. Combe, and V. Conan, "Cost-based placement of vdpi functions in nfv infrastructures," *International Journal in Network Management*, vol. 25, pp. 490–506, November 2015.

[CCW+12] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, C. Cui, H. Deng, J. Benitez, U. Michel, H. Damker, K. Ogaki, T. Matsuzaki, M. Fukui, K. Shimano, D. Delisle, Q. Loudier, C. Kolias, I. Guardini, E. Demaaria, R. Minerva, A. Manzalini, D. López, F. J. R. Salguero, F. Ruhl, and P. Sen, "White paper: Network

functions virtualization," AT&T, BT, CenturyLink, China Mobile, Colt, Deutsche Telekom, KDDI, NTT, Orange, Telecom Italia, Telefonica, Telstra, Verizon, Darmstad, Germany, Tech. Rep., Oct. 2012. [Online]. Available: http://portal.etsi.org/NFV/NFV_White_Paper.pdf

[CFA+07] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *CGO '07 Proceedings of the International Symposium on Code Generation and Optimization*, 2007, pp. 185–197.

[CM05] G. Contreras and M. Martonosi, "Power prediction for intel xscale®processors using performance monitoring unit events," in *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ser. ISLPED '05. New York, NY, USA: ACM, 2005, pp. 221–226. [Online]. Available: http://doi.acm.org/10.1145/1077603.1077657

[EEKS06] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate cpi components," in *ASPLOS XII Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006, pp. 175–184.

[ETS13] ETSI, *Network Functions Virtualisation (NFV); Use Cases*, European Telecommunications Standards Institute (ETSI) Std., Rev. 1.1.1, Oct. 2013. [Online]. Available: http://www.etsi.org/deliver/etsi_gs/NFV/001_099/001/01.01.01_60/gs_NFV001v010101p.pdf

[ETS14] ——, *Network Functions Virtualisation (NFV); Virtual Network Functions Architecture*, European Telecommunications Standards Institute (ETSI) Std., Rev. 1.1.1, Dec. 2014. [Online]. Available: http://www.etsi.org/deliver/etsi_gs/NFV-SWA/001_099/001/01.01.01_60/gs_NFV-SWA001v010101p.pdf

[FKLM13] L. Fahrmeier, T. Kneib, S. Lang, and B. Marx, *Regression Models, Methods and Applications*. Springer Berlin Heidelberg, 2013.

[GR] R. Gooch and L. R. Rodriguez, "Mtrr (memory type range register) control," accessed: 2016-08-15. [Online]. Available: http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/x86/mtrr.txt

[HJJ03] D. Henderson, S. H. Jacobson, and A. W. Johnson, *The Theory and Practice of Simulated Annealing*, F. Glover and G. A. Kochenberger, Eds. Boston, MA: Springer US, 2003. [Online]. Available: http://dx.doi.org/10.1007/0-306-48056-5_10

[IM03] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 93–. [Online]. Available: http://dl.acm.org/citation.cfm?id=956417.956567

[Int] Intel, "Intel® Xeon® Processor E3-1230 v5 (8M Cache, 3.40 GHz)," accessed: 08.09.2016. [Online]. Available: http://ark.intel.com/products/88182/Intel-Xeon-Processor-E3-1230-v5-8M-Cache-3_40-GHz

[Int16a] ——, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, June 2016.

[Int16b] ——, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, June 2016.

[JWC12] Y. Jin, Y. Wen, and Q. Chen, "Energy efficiency and server virtualization in data centers: An empirical investigation," in *2012 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Mar. 2012, pp. 133–138.

[KTK+01] I. Kadayif, T.Chinoda, M. Kandemir, N. Vijaykrishnan, M. Irwin, and A. Sivasubramaniam, "vec: Virtual energy counters," in *PASTE'01 Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001, pp. 28–31.

[LAB+13] K.-D. Lange, J. A. Arnold, H. Block, N. Totura, J. Beckett, and M. G. Tricker, "Further implementation aspects of the server efficiency rating tool (sert)," in *4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, April 2013.

[LGT08] A. Lewis, S. Ghosh, and N.-F. Tzeng, "Runtime energy consumption estimation based on workload in server systems," in *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, ser. HotPower'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 4–4. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855610.1855614

[LT11] K.-D. Lange and M. G. Tricker, "The Design and Development of the Server Efficiency Rating Tool (SERT)," in *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '11. New York, NY, USA: ACM, 2011, pp. 145–150. [Online]. Available: http://doi.acm.org/10.1145/1958746.1958769

[LTA+12] K.-D. Lange, M. G. Tricker, J. A. Arnold, H. Block, and C. Koopmann, "The implementation of the server efficiency rating tool," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '12. New York, NY, USA: ACM, 2012, pp. 133–144. [Online]. Available: http://doi.acm.org/10.1145/2188286.2188307

[MDT14] H. Moens and F. De Turck, "Vnf-p: A model for efficient placement of virtualized network functions," in *10e International Conference on Network and Service Management, Proceedings*, 2014, pp. 418–423.

[MSG+15] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. D. Turck, and S. Davy, "Design and evaluation of algorithms for mapping and scheduling of virtual network functions," in *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. IEEE, Apr. 2015, pp. 1–9.

[Ols05] R. Olsson, "pktgen the linux packet generator," in *Proceedings of the Linux Symposium*, vol. 2, July 2005.

[OPN] OPNFV, "Opnfv technical overview," https://www.opnfv.org/software/technical-overview, accessed: 29.03.2016.

[PNV+10] M. Poess, R. O. Nambiar, K. Vaid, J. M. S. Jr, K. Huppler, and E. Haines, "Energy benchmarks: A detailed analysis," in *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*. ACM, 2010.

[Pro06] T. L. I. Project, "Software interrupt definition," April 2006, accessed: 21.09.2016. [Online]. Available: http://www.linfo.org/software_interrupt.html

[SAP11] SAP, "Sap power benchmarks specification," http://global.sap.com/solutions/benchmark/pdf/Specification_SAP_Power_Benchmarks_V12.pdf, February 2011, accessed: 28.09.2016.

[SBM09] K. Singh, M. Bhadauria, and S. A. McKee, "Real time power estimation and thread scheduling via performance counters," *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 46–55, May 2009.

[SPEa] SPEC, *Chauffeur^{TM} Worklet Development Kit (WDK) User Guide 1.1.0*, 7001 Heritage Village Plaza, Suite 225 Gainesville, VA 20155, USA.

[SPEb] ——, *Power and Performance Benchmark Methodology V2.2*, Standard Performance Evaluation Corporation (SPEC), 7001 Heritage Village Plaza, Suite 225 Gainesville, VA 20155, USA.

[SPE13] ——, "Server efficiency rating tool (sert) design document 1.0.2," https://www.spec.org/sert/docs/SERT-Design_Doc.pdf, 2013.

[vKBB+15] J. v. Kistowski, H. Block, J. Beckett, K.-D. Lange, J. A. Arnold, and S. Kounev, "Analysis of the influences on server power consumption and energy efficiency for cpu-intensive workloads," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15. New York, NY, USA: ACM, 2015, pp. 223–234. [Online]. Available: http://doi.acm.org/10.1145/2668930.2688057

[vKBB+16] J. v. Kistowski, H. Block, J. Beckett, C. Spradling, K.-D. Lange, and S. Kounev, "Variations in cpu power consumption," in *ICPE'16 Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, 2016, pp. 147–158.

[vKBL+15] J. v. Kistowski, J. Beckett, K.-D. Lange, H. Block, J. A. Arnold, and S. Kounev, "Energy efficiency of hierarchical server load distribution strategies," in *IEEE 23rd International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, October 2015.

[vKHK14] J. G. von Kistowski, N. R. Herbst, and S. Kounev, "Modeling Variations in Load Intensity over Time," in *Proceedings of the 3rd International Workshop on Large-Scale Testing (LT 2014), co-located with the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*. New York, NY, USA: ACM, Mar. 2014, pp. 1–4. [Online]. Available: http://doi.acm.org/10.1145/2577036.2577037

[WD14] J. Whitney and P. Delforge, "Data center efficiency assessment," http://www.nrdc.org/energy/files/data-center-efficiency-assessment-IP.pdf, Aug. 2014.

[Wea15] V. M. Weaver, "Self-monitoring overhead of the linux perf_event performance counter interface," in *Performance Analysis of Systems and Software, 2015. ISPASS 2015. IEEE International Symposium on*. IEEE, March 2015, pp. 102–111.

[Wil] T. Willhalm, "Intel pcm column names decoder ring," accessed: 15.09.2016. [Online]. Available: https://software.intel.com/en-us/blogs/2014/07/18/intel-pcm-column-names-decoder-ring

[Wol16] J. Wolf, *Linux-UNIX-Programmierung: Das umfassende Handbuch*, ser. Rheinwerk Computing. Rheinwerk Verlag GmbH, 2016.

[WTM13] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *Performance Analysis of Systems and Software, 2013. ISPASS 2013. IEEE International Symposium on*, 2013.

[ZJH09] D. Zaparanuks, M. Jovic, and M. Hauswirth, "Accuracy of performance counter measurements," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, 2009, pp. 23 –32.

List of Figures

List of Tables

Appendix

A. SUT Background Noise

	Performance counter	System	Socket	Core	Base value
IntelPCM	L3 misses			✓	941.06 ± 65.00
	L2 misses			✓	$(13.000 \pm 0.851) \cdot 10^3$
	L3 hit ratio			✓	0.9152 ± 0.0030
	L2 hit ratio			✓	0.2763 ± 0.0057
	L3 hits (from hit ratio)			✓	1028.97 ± 72.02
	L2 hits (from hit ratio)			✓	$(46.658 \pm 1.868) \cdot 10^3$
	Bytes read		✓		$(55.260 \pm 60.365) \cdot 10^3$
	Bytes written		✓		$(19.779 \pm 23.377) \cdot 10^3$
	Instructions retired	✓			$(31.93 \pm 2.23) \cdot 10^6$
Linux	Interrupts	✓			22.87 ± 1.04
	Context switches	✓			30.73 ± 1.97

Table A.1.: *SUT system wide background noise for* 1 s *averaged over* 120 s *with* 95 % *confidence interval*

B. Testbed Hardware

B.1. SUT

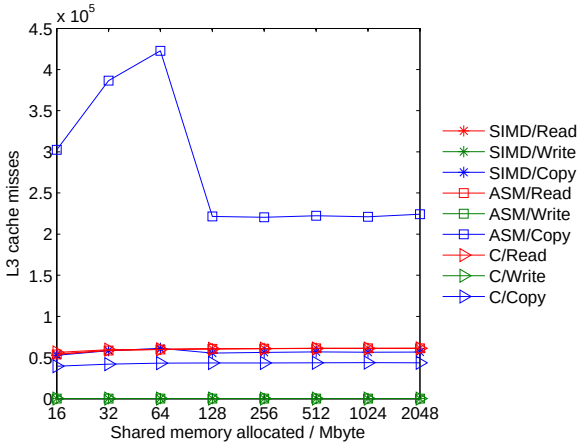
- **CPU**: Intel Xeon E3-1230 v5
  - 4 cores with 2 threads per core
  - 3.4 GHz and 3.8 GHz with turbo
  - 32 KiB L1 data cache per core
  - 32 KiB L1 instruction cache per core
  - 256 KiB L2 shared cache per core

  - 8 MiB L3 shared cache
  - 64 byte cache line size
- **Memory**
  - 16 GiB total capacity (1 module)
  - Single channel
  - 2133 MHz clock speed
- **NIC 1**: HPE Ethernet 1Gb 2-port 332i Adapter
  - Dual-Port 10/100/1000Base-T full-duplex/half-duplex
  - Broadcom NetXtreme BCM5720
- **NIC 2**: HP Ethernet 1Gb 2-port 332T Adapter
  - Dual-Port 10/100/1000Base-T full-duplex/half-duplex
  - Broadcom NetXtreme BCM5720
  - 2.2 W maximum power consumption
  - 2.1 W idle power consumption

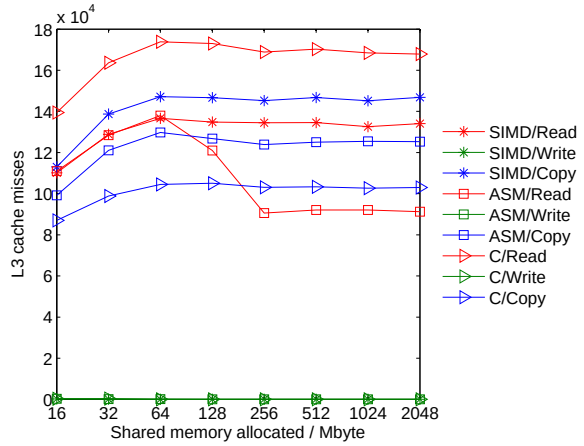
B.2. Traffic Generator and Receiver

- **CPU**: Intel Xeon E5-2560 v3
  - 2.3 GHz and 3.0 GHz with turbo
  - 32 KiB L1 data cache per core
  - 32 KiB L1 instruction cache per core
  - 256 KiB L2 shared cache per core
  - 25 MiB L3 shared cache
  - 64 byte cache line size
- **Memory**
  - 32 GiB total capacity (2 modules, each 16 GiB)
  - Dual channel
  - 2133 MHz clock speed

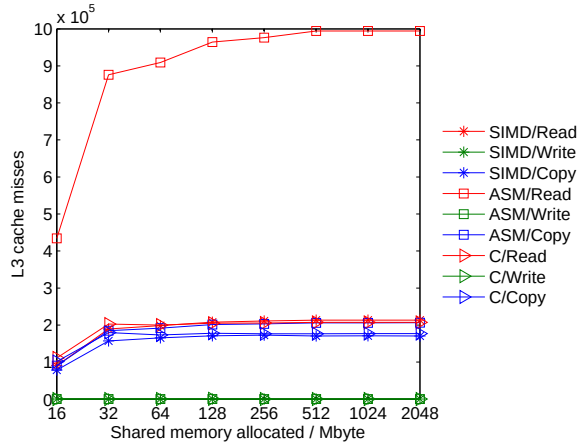
C. Measurement Results of Selected Performance Counter



(a) Step size 2

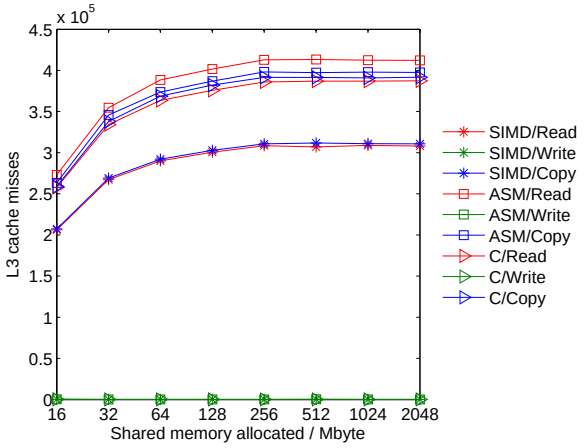


(b) Step size 4

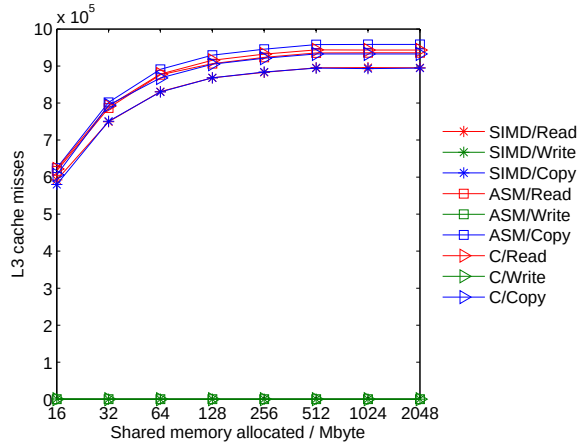


(c) Step size 6

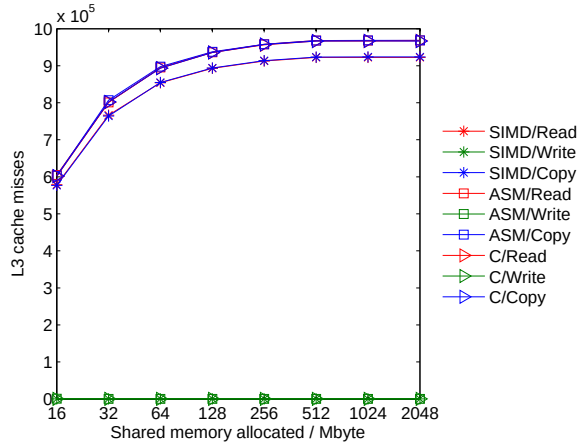
Figure C.1.: *L3 cache misses for one process, shared memory and no random factor*



(a) Step size 2

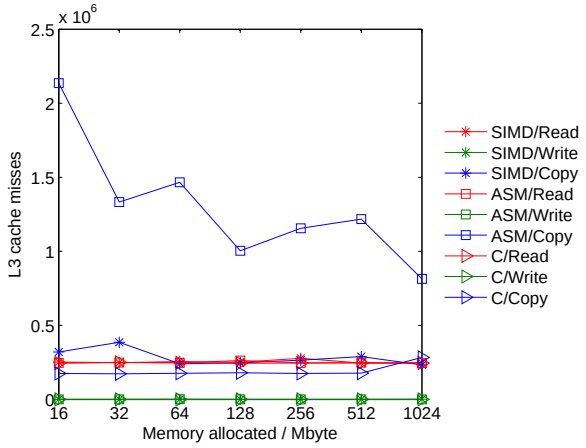


(b) Step size 4

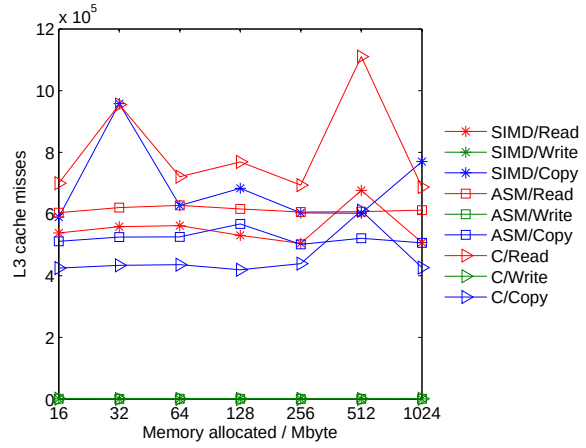


(c) Step size 6

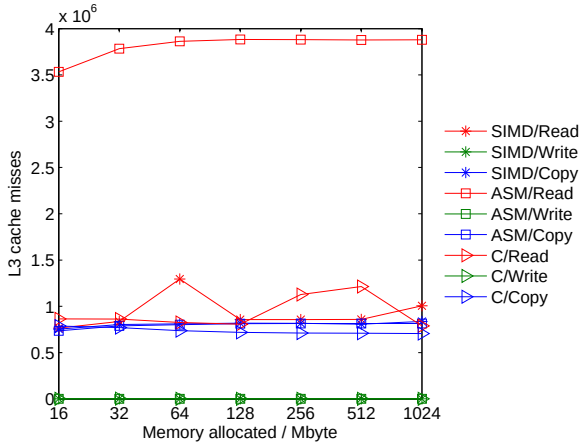
Figure C.2.: *L3 cache misses for one process and shared memory with random factor*



(a) Step size 2

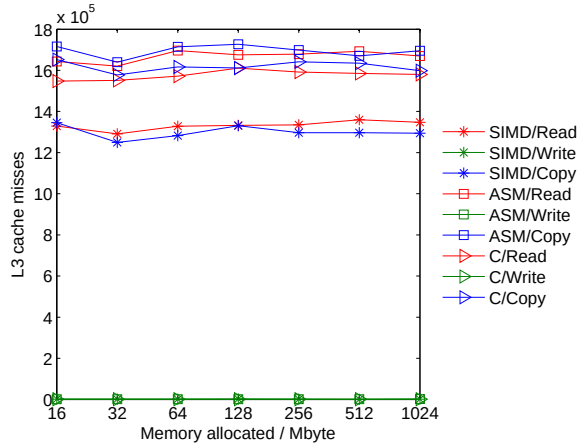


(b) Step size 4

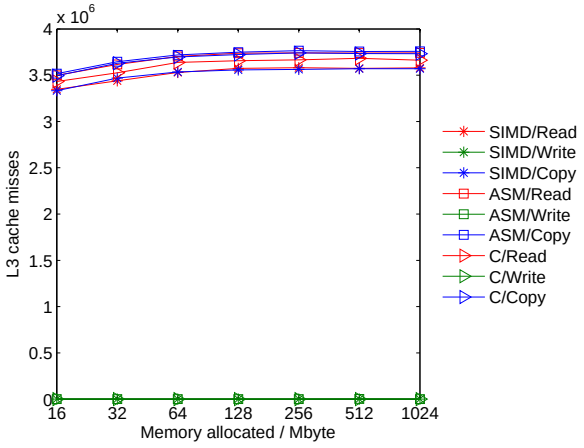


(c) Step size 6

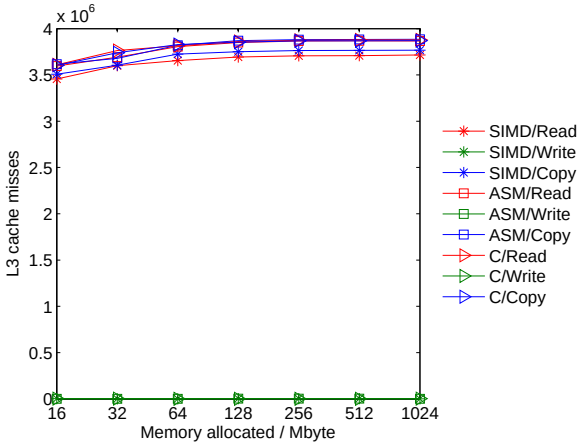
Figure C.3.: *L3 cache misses for four processes, process virtual memory and no random factor*



(a) Step size 2

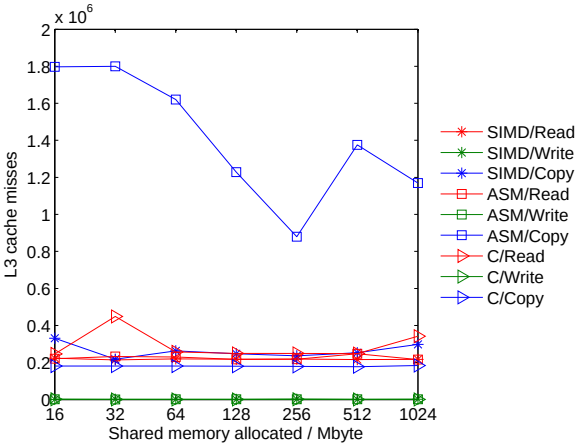


(b) Step size 4

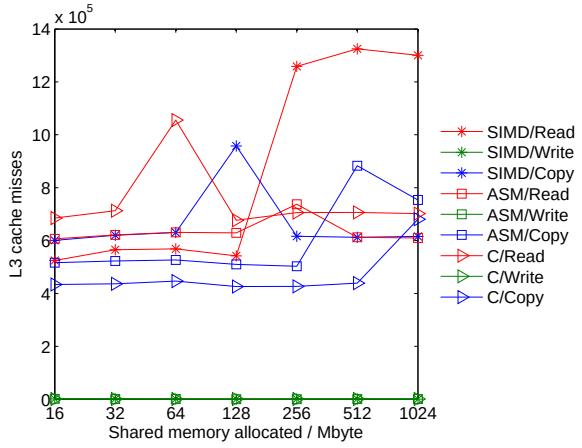


(c) Step size 6

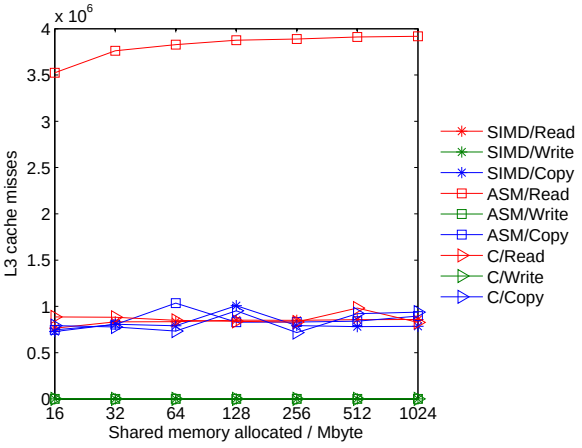
Figure C.4.: *L3 cache misses for four processes and process virtual memory with random factor*



(a) Step size 2

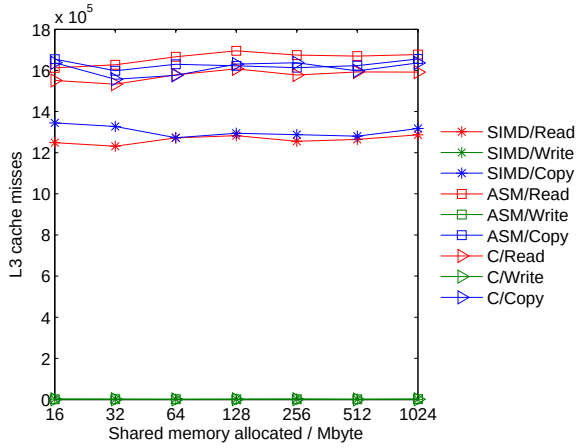


(b) Step size 4

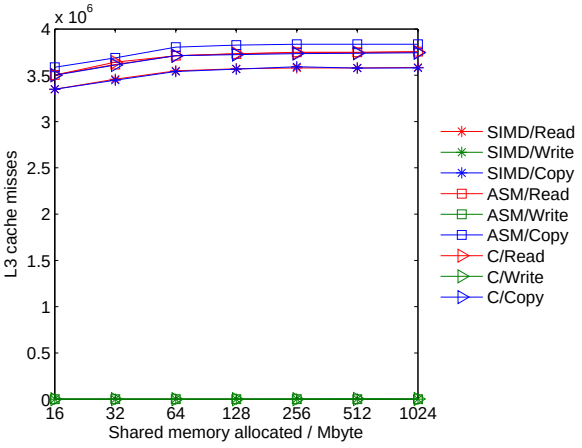


(c) Step size 6

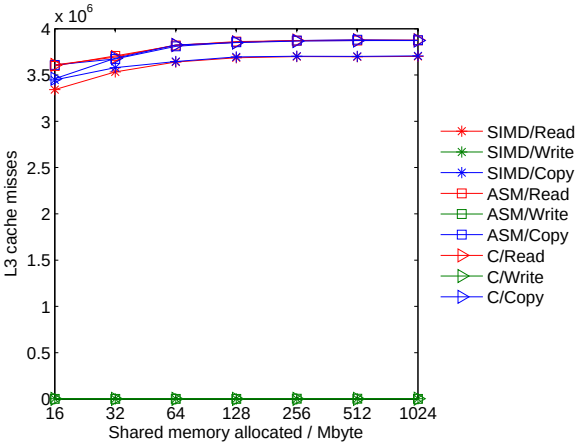
Figure C.5.: *L3 cache misses for four processes, shared memory and no random factor*



(a) Step size 2



(b) Step size 4



(c) Step size 6

Figure C.6.: *L3 cache misses for four processes and shared memory with random factor*

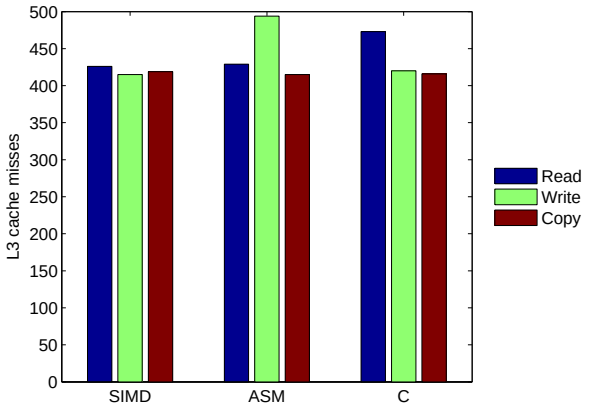
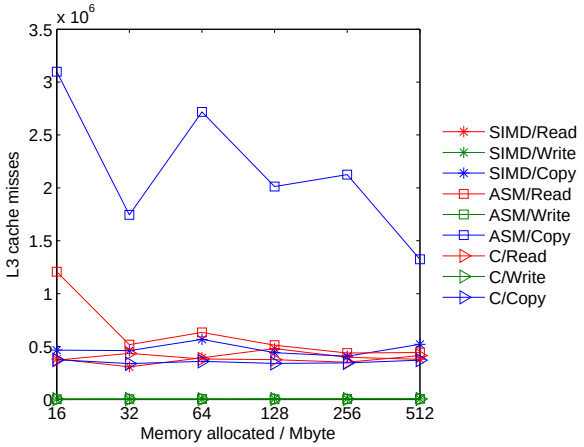
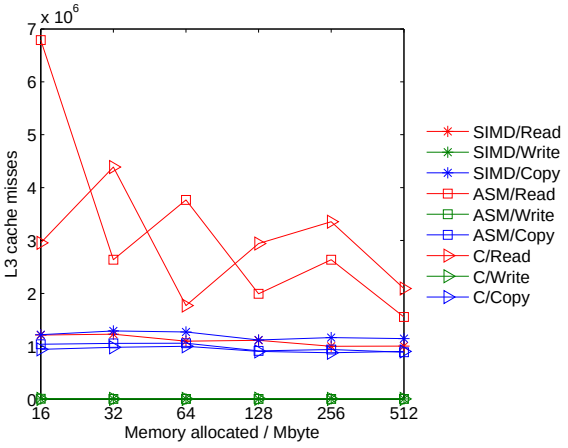


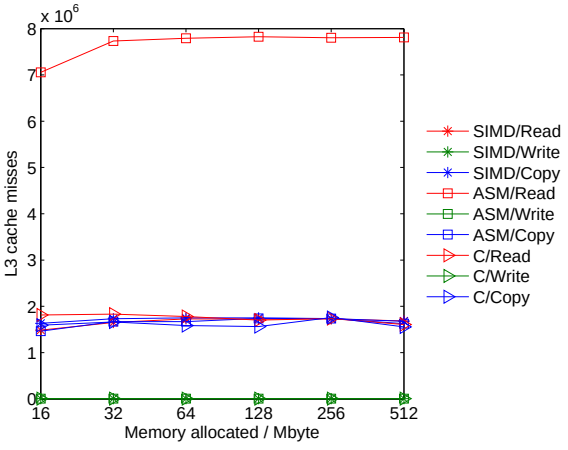
Figure C.7.: *L3 cache miss results for four processes and UC- memory with fixed array indices*



(a) Step size 2

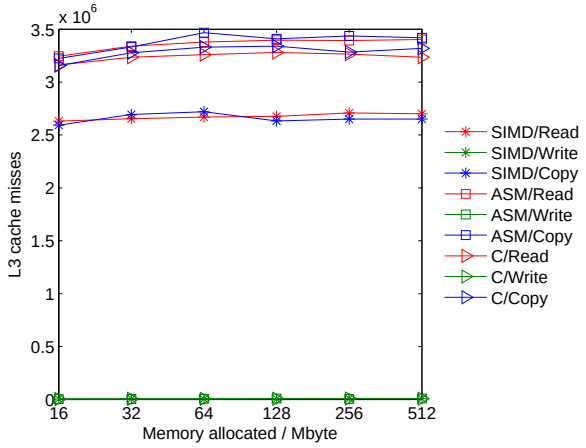


(b) Step size 4

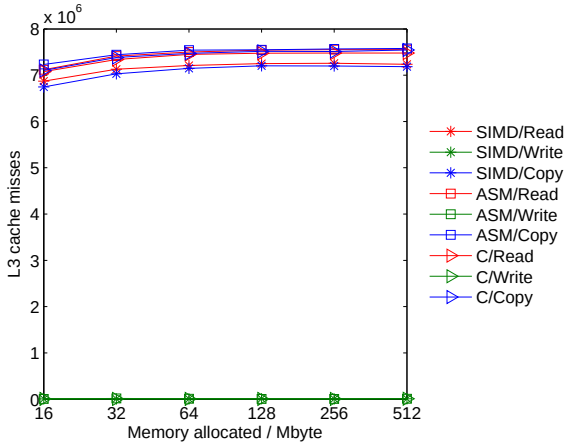


(c) Step size 6

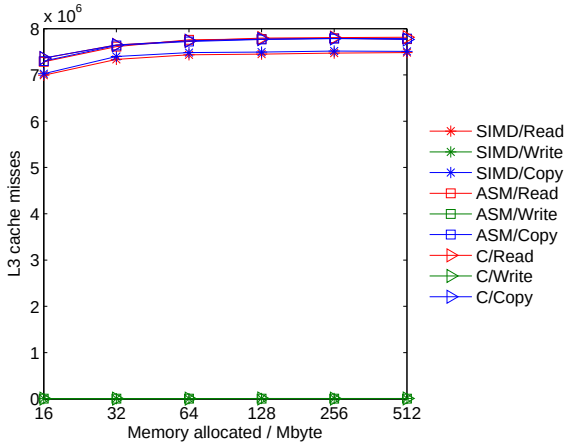
Figure C.8.: *L3 cache misses for eight processes, process virtual memory and no random factor*



(a) Step size 2

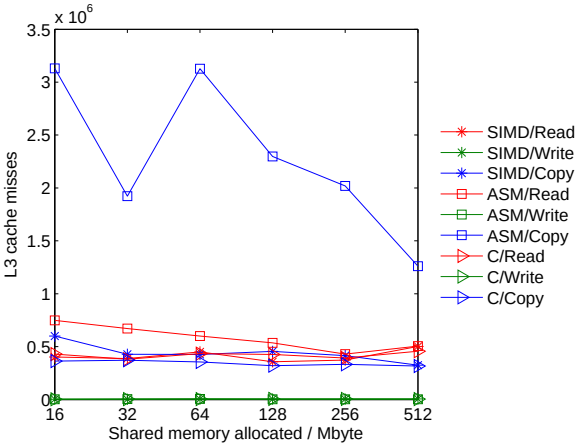


(b) Step size 4

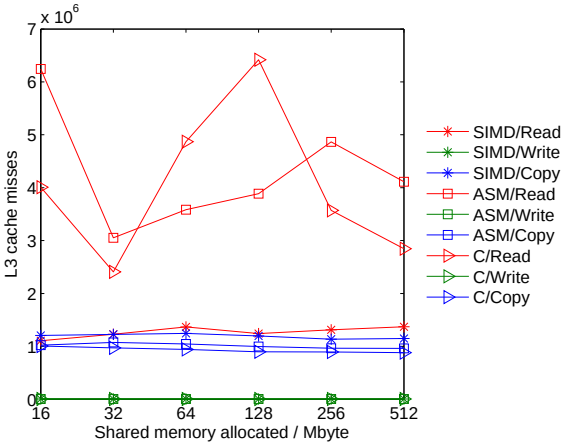


(c) Step size 6

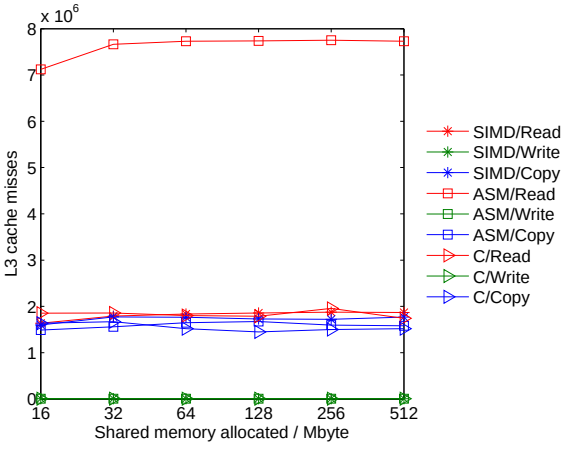
Figure C.9.: *L3 cache misses for eight processes and process virtual memory with random factor*



(a) Step size 2

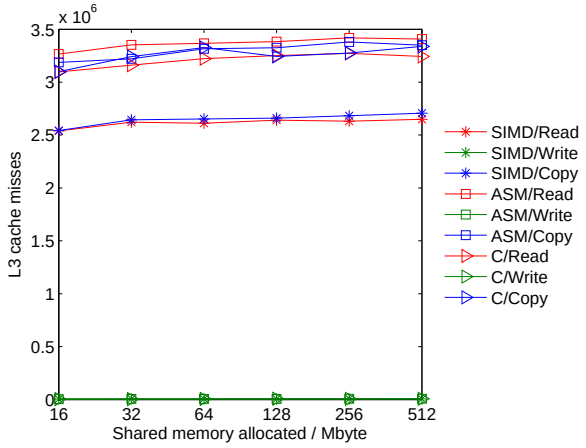


(b) Step size 4

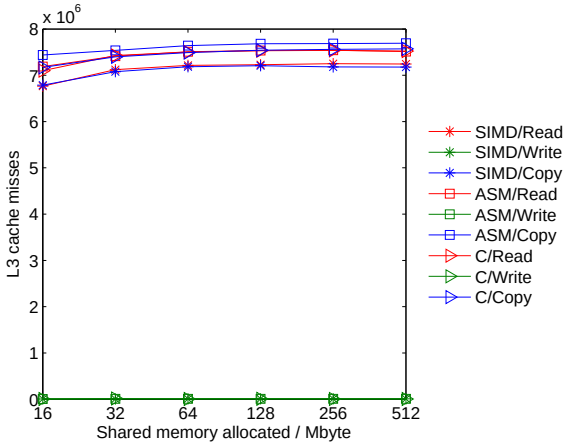


(c) Step size 6

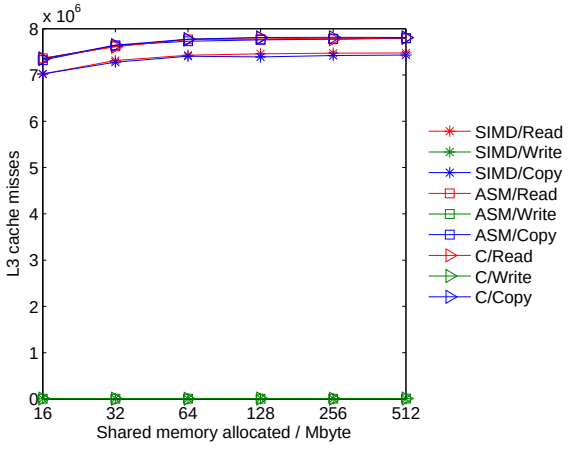
Figure C.10.: *L3 cache misses for eight processes, shared memory and no random factor*



(a) Step size 2



(b) Step size 4



(c) Step size 6

Figure C.11.: *L3 cache misses for eight processes and shared memory with random factor*

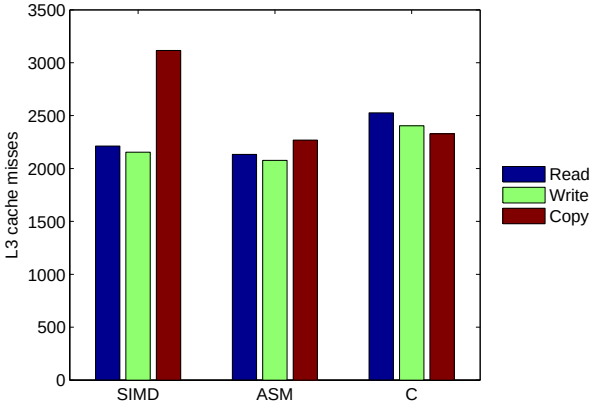
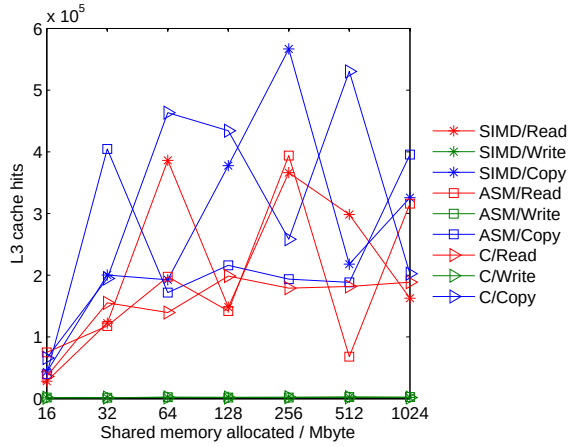
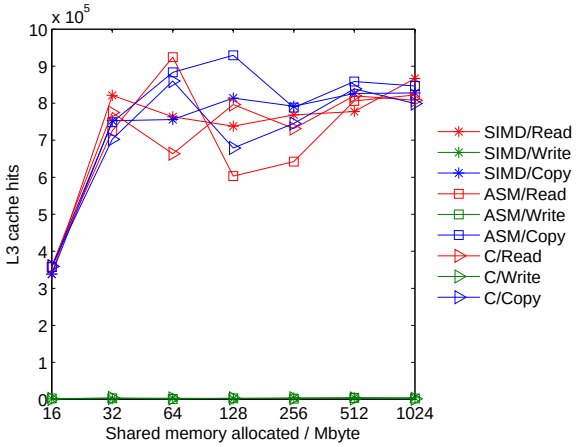


Figure C.12.: *L3 cache misses for eight processes and UC- memory with fixed array indices*

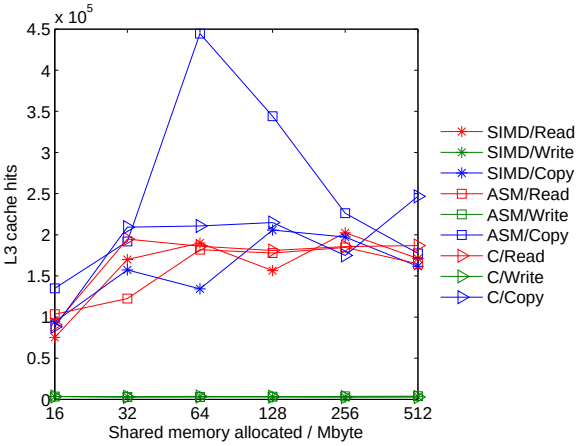


(a) Without random factor

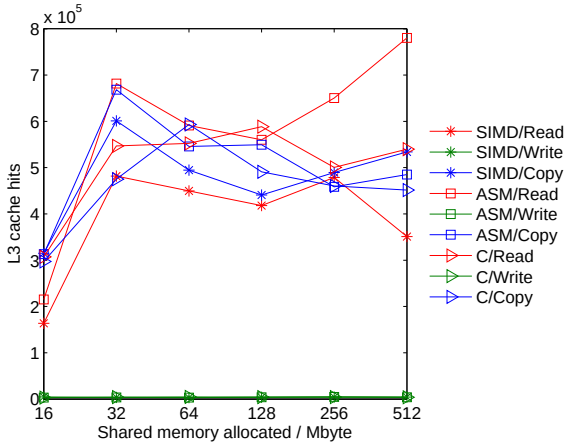


(b) With random factor

Figure C.13.: *L3 cache hits for four processes using shared memory*



(a) Without random factor



(b) With random factor

Figure C.14.: *L3 cache hits for eight processes using shared memory*

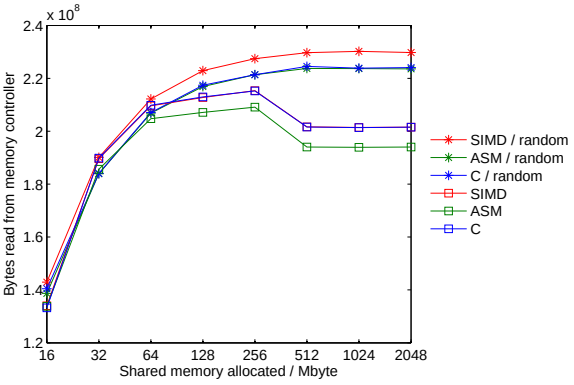
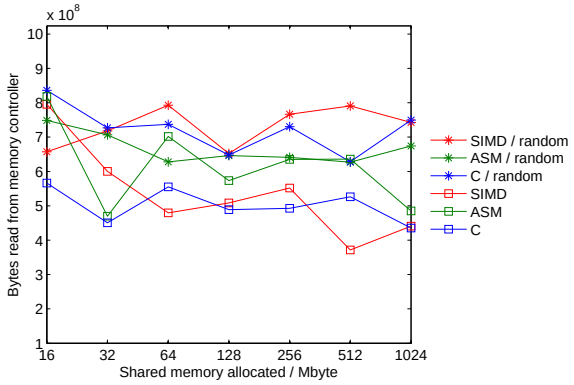
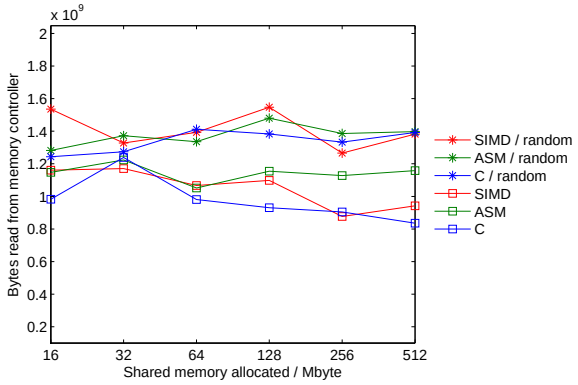


Figure C.15.: *Bytes read for one process using shared memory*



(a) 4 processes



(b) 8 processes

Figure C.16.: *Bytes read for four and eight processes using shared memory*

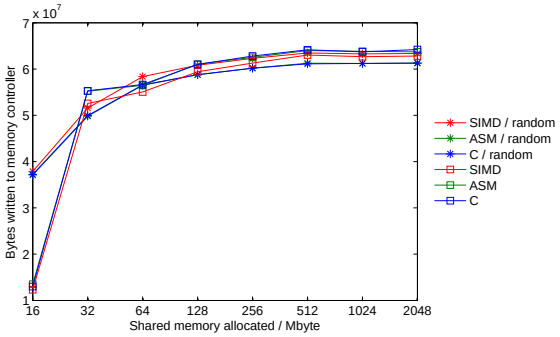
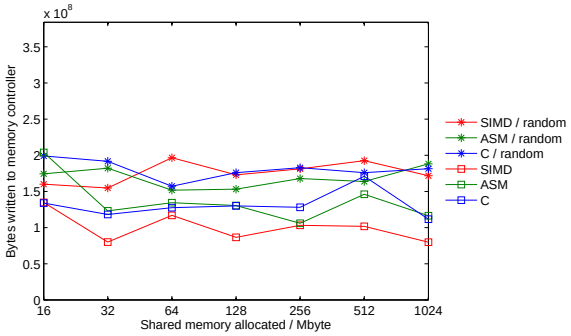
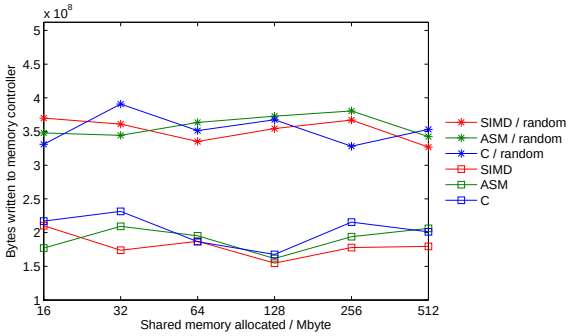


Figure C.17.: *Bytes written for one process using shared memory*



(a) 4 processes



(b) 8 processes

Figure C.18.: *Bytes written for four and eight processes using shared memory*

D. PET Side Effect Configuration

Processes	Performance counter	L3 cache hits	L3 cache misses	Bytes read	Bytes written	Instructions retired	Interrupts	Context switches
1	L3 cache hits		$1.6 \cdot 10^{-3}$	$0.7 \cdot 10^{-3}$	$1.2 \cdot 10^{-3}$	$2 \cdot 10^{-7}$	0.224	46
	L3 cache misses	$2.6 \cdot 10^{-3}$		$0.03 \cdot 10^{-3}$	$0.4 \cdot 10^{-3}$	$0.14 \cdot 10^{-7}$	0.064	7
	Bytes read	0.6	195		192	$1.7 \cdot 10^{-5}$	51	5263
	Bytes written	0.4	3	0.014		$1 \cdot 10^{-5}$	26	4271
	Instructions retired	74	129	22	129		15 259	23 743
	Interrupts	$1 \cdot 10^{-5}$	$2 \cdot 10^{-5}$	$3 \cdot 10^{-5}$	$1 \cdot 10^{-5}$	$3 \cdot 10^{-8}$		$4.5 \cdot 10^{-3}$
	Context switches	$17 \cdot 10^{-5}$	$17 \cdot 10^{-5}$	$17 \cdot 10^{-5}$	$17 \cdot 10^{-5}$	$8 \cdot 10^{-8}$	2	
4	L3 cache hits		$2.5 \cdot 10^{-3}$	$0.9 \cdot 10^{-3}$	$3.2 \cdot 10^{-3}$	$5 \cdot 10^{-7}$	0.41	49
	L3 cache misses	$50 \cdot 10^{-3}$		$2 \cdot 10^{-3}$	$1.2 \cdot 10^{-3}$	$0.49 \cdot 10^{-7}$	0.142	10
	Bytes read	50	98		122	$12 \cdot 10^{-5}$	80	7177
	Bytes written	1.8	4	0.002		$31 \cdot 10^{-5}$	24	4624
	Instructions retired	75	129	22	130		15 321	23 572
	Interrupts	$7 \cdot 10^{-5}$	$2 \cdot 10^{-5}$	$3 \cdot 10^{-5}$	$2 \cdot 10^{-5}$	$5 \cdot 10^{-8}$		$2.8 \cdot 10^{-3}$
	Context switches	$56 \cdot 10^{-5}$	$14 \cdot 10^{-5}$	$16 \cdot 10^{-5}$	$93 \cdot 10^{-5}$	$9 \cdot 10^{-8}$	2	
8	L3 cache hits		$4.3 \cdot 10^{-3}$	$3.6 \cdot 10^{-3}$	$4 \cdot 10^{-3}$	$6 \cdot 10^{-7}$	4.637	74
	L3 cache misses	$40 \cdot 10^{-3}$		$4.3 \cdot 10^{-3}$	$0.8 \cdot 10^{-3}$	$1 \cdot 10^{-7}$	0.341	10
	Bytes read	38	85		64	$20 \cdot 10^{-5}$	265	7605
	Bytes written	10	21	13.6		$56 \cdot 10^{-5}$	119	4732
	Instructions retired	75	131	24	131		15 385	23 620
	Interrupts	$25 \cdot 10^{-5}$	$3 \cdot 10^{-5}$	$3 \cdot 10^{-5}$	$2 \cdot 10^{-5}$	$5 \cdot 10^{-8}$		$10.5 \cdot 10^{-3}$
	Context switches	$71 \cdot 10^{-5}$	$22.0 \cdot 10^{-5}$	$8.5 \cdot 10^{-5}$	$186 \cdot 10^{-5}$	$12 \cdot 10^{-8}$	2	

Table D.2.: *PET side effect configuration per event generated*

E. Measurement Results of PET

			100 %	90 %	80 %	70 %	60 %	50 %	40 %	30 %	20 %	10 %
Pi	Full	Naive	0.0	1.5	1.7	1.1	1.2	1.8	1.2	2.3	2.7	4.1
		Accumulation	0.3	0.9	1.4	1.2	0.9	1.2	1.3	2.3	3.0	4.0
		Simulated Annealing	0.5	2.8	2.9	5.5	5.2	4.0	8.3	6.4	8.2	12.2
	Reduced	Naive	0.5	0.1	0.1	0.2	0.2	0.2	0.3	0.2	0.4	0.4
		Accumulation	0.2	0.1	0.2	0.1	0.1	0.1	0.1	0.2	0.2	0.2
		Simulated Annealing	1.3	0.2	0.1	0.2	0.2	0.3	0.2	0.3	0.3	0.6
XMLVal.	Full	Naive	28.0	11.1	9.8	8.0	11.0	11.8	13.1	17.3	21.7	18.0
		Accumulation	0.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1
		Simulated Annealing	0.4	1.8	1.9	3.6	3.8	4.5	1.9	5.6	6.4	11.4
	Reduced	Naive	28.2	13.1	9.7	9.1	10.0	13.0	8.0	12.8	20.2	18.5
		Accumulation	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1
		Simulated Annealing	0.5	2.2	2.1	1.6	3.3	2.5	3.3	3.1	3.6	6.2
SSJ	Full	Naive	33.5	20.3	12.6	7.8	6.6	10.7	7.9	4.8	9.6	18.6
		Accumulation	0.1	0.1	0.1	0.1	0.1	0.1	0.2	0.2	0.2	0.3
		Simulated Annealing	0.1	0.8	1.0	0.6	0.5	0.9	0.8	1.5	0.9	3.4
	Reduced	Naive	33.2	18.9	13.4	6.7	1.0	0.8	0.9	1.5	1.5	0.8
		Accumulation	1.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1
		Simulated Annealing	0.2	0.1	0.1	0.1	0.1	0.1	0.1	0.0	0.2	0.2
NFV	Full	Naive	11.1	3.5	0.7	1.4	2.1	1.4	1.2	1.0	2.8	4.3
		Accumulation	0.2	0.3	0.3	0.5	0.4	0.4	0.4	0.8	0.5	0.9
		Simulated Annealing	0.2	0.6	0.5	0.5	0.2	0.6	1.0	1.0	1.5	1.3
	Reduced	Naive	21.6	4.7	2.2	1.8	1.4	1.6	1.2	1.8	2.5	4.2
		Accumulation	0.7	0.5	0.3	0.5	0.4	0.2	0.5	0.6	0.7	1.4
		Simulated Annealing	0.1	0.6	0.7	0.4	0.5	0.3	0.7	1.2	1.3	1.5
	Elongated	Naive	12.7	2.4	1.3	0.7	0.8	0.8	0.8	0.7	1.0	2.4
		Accumulation	0.6	0.2	0.3	0.3	0.3	0.4	0.4	0.5	0.5	1.0
		Simulated Annealing	0.3	0.2	0.4	0.5	0.4	0.4	0.5	0.8	0.7	1.2

Table E.3.: *PET measurement client CVs in percent*

			100 %	90 %	80 %	70 %	60 %
Pi	Full	Naive	380.809	343.990	303.610	266.520	229.367
		Accumulation	479.737	438.357	387.340	343.340	294.157
		Simulated Annealing	52.826	47.580	42.001	36.287	31.922
	Reduced	Naive	36 012.081	32 351.458	28 760.722	25 182.340	21 561.546
		Accumulation	40 606.948	36 535.929	32 509.667	28 424.921	24 343.007
		Simulated Annealing	21 112.418	19 021.441	16 904.164	14 798.229	12 690.113
XMLValidate	Full	Naive	8.871	7.288	6.489	6.014	5.115
		Accumulation	1 157 426.419	1 006 465.600	894 490.805	782 745.217	670 960.849
		Simulated Annealing	60.469	56.234	49.324	44.483	37.427
	Reduced	Naive	8.912	7.247	6.839	6.090	5.356
		Accumulation	1 167 237.733	1 006 371.118	894 513.505	782 694.587	670 917.963
		Simulated Annealing	154.038	137.064	124.103	109.116	91.931
SSJ	Full	Naive	18.075	14.402	12.904	11.896	11.071
		Accumulation	61 553.056	55 282.444	49 169.292	43 051.547	36 859.142
		Simulated Annealing	1409.036	1271.732	1130.232	986.266	852.100
	Reduced	Naive	1807.456	1421.734	1312.344	1198.032	1091.188
		Accumulation	1 464 869.960	1 006 438.538	894 629.032	782 725.412	670 892.260
		Simulated Annealing	139 138.535	125 274.615	111 304.292	97 399.069	83 502.123
NFV	Full	Naive	568.159	513.253	468.480	410.657	352.259
		Accumulation	6261.788	5631.564	5003.889	4382.502	3755.880
		Simulated Annealing	2351.546	2098.054	1862.708	1635.208	1395.707
	Reduced	Naive	615.403	499.126	455.790	402.607	341.956
		Accumulation	6224.015	5590.007	4969.151	3730.289	3108.501
		Simulated Annealing	2360.878	2130.968	1898.705	1663.466	1423.847
	Elongated, Full	Naive	576.179	518.241	465.187	410.743	351.810
		Accumulation	6234.202	5610.808	4989.049	4361.637	3740.267
		Simulated Annealing	2342.601	2104.259	1867.088	1640.858	1402.155

Table E.4.: *PET measurement score for the load levels* 100 % *to* 60 %

122

			50 %	40 %	30 %	20 %	10 %
Pi	Full	Naive	190.184	154.273	114.025	75.490	38.070
		Accumulation	242.839	198.037	147.358	96.517	47.383
		Simulated Annealing	26.839	21.742	16.261	10.737	4.932
	Reduced	Naive	17 972.323	14 394.429	10 798.924	7186.843	3607.260
		Accumulation	20 290.226	16 243.435	12 187.060	8113.700	4062.354
		Simulated Annealing	10 562.533	8472.825	6345.579	4232.016	2109.637
XMLValidate	Full	Naive	3.999	3.349	2.616	1.866	0.875
		Accumulation	558 948.090	447 365.597	335 518.846	223 639.201	111 800.429
		Simulated Annealing	30.488	24.874	17.735	12.804	6.389
	Reduced	Naive	4.748	3.257	2.574	1.858	0.841
		Accumulation	559 058.543	447 139.916	335 469.400	223 609.678	111 832.315
		Simulated Annealing	75.506	60.496	46.243	30.838	15.003
SSJ	Full	Naive	9.122	7.672	5.332	3.640	2.058
		Accumulation	30 752.249	24 586.605	18 463.804	12 304.054	6147.057
		Simulated Annealing	704.434	561.562	422.868	282.000	140.176
	Reduced	Naive	908.105	721.652	545.409	361.647	181.013
		Accumulation	559 034.766	447 207.007	335 433.629	223 611.791	111 841.141
		Simulated Annealing	69 537.406	55 616.628	41 766.498	27 859.051	13 911.666
NFV	Full	Naive	295.258	235.364	176.278	117.938	57.596
		Accumulation	3133.878	2506.927	1875.712	1248.417	623.110
		Simulated Annealing	1160.421	934.384	700.266	466.862	236.022
	Reduced	Naive	286.188	232.140	172.354	114.990	57.988
		Accumulation	3108.501	2488.278	1868.550	1240.955	620.770
		Simulated Annealing	1183.651	947.338	712.783	470.562	236.882
	Elongated, Full	Naive	294.658	237.577	176.666	117.666	59.008
		Accumulation	3117.168	2496.720	1865.586	1244.295	623.201
		Simulated Annealing	1170.272	937.236	701.542	467.769	233.682

Table E.5.: *PET measurement score for the load levels* 50 % *to* 10 %

F. Linear Regression Model

Performance counter	Estimated coefficient β	Standard Error
Intercept	23.185	1.1963
L3 misses	$9.7313 \cdot 10^{-5}$	$1.6844 \cdot 10^{-5}$
L3 hits	$-5.939 \cdot 10^{-5}$	$1.0333 \cdot 10^{-5}$
Bytes read from memory controller	$-5.304 \cdot 10^{-8}$	$9.6052 \cdot 10^{-9}$
Instructions retired	$7.4655 \cdot 10^{-9}$	$1.0876 \cdot 10^{-9}$
Interrupts	$1.6188 \cdot 10^{-3}$	$7.3213 \cdot 10^{-4}$
Context switches	$-9.3015 \cdot 10^{-4}$	$2.0541 \cdot 10^{-4}$

Table F.6.: *Estimated coefficients β of linear regression model with not significant counters removed*

	Workload			
Load Level	Pi	XMLValidate	SSJ	NFV
100 %	−1.38	−400.16	−494.54	−371.19
90 %	−0.38	−348.51	−394.95	−341.78
80 %	1.85	−308.33	−337.54	−306.64
70 %	4.17	−270.68	−288.19	−271.30
60 %	6.64	−232.69	−244.03	−231.06
50 %	9.41	−193.58	−192.65	−188.99
40 %	12.67	−154.56	−145.94	−147.73
30 %	15.92	−116.66	−103.75	−104.64
20 %	18.14	−78.02	−54.40	−61.47
10 %	20.52	−40.90	−8.66	−18.91

Table F.7.: *Power consumption prediction of the linear regression model in watts with insignificant coefficients removed*

Performance counter	Estimated coefficient β	Standard Error
Intercept	30.807	2.8458
L3 misses	$1.1915 \cdot 10^{-5}$	$6.6617 \cdot 10^{-5}$
L3 hits	$-7.6529 \cdot 10^{-6}$	$4.2786 \cdot 10^{-5}$
Bytes read from memory controller	$-9.0466 \cdot 10^{-10}$	$2.9851 \cdot 10^{-8}$
Bytes written to memory controller	$-1.7218 \cdot 10^{-8}$	$3.3599 \cdot 10^{-8}$
Instructions retired	$3.6391 \cdot 10^{-9}$	$5.2868 \cdot 10^{-9}$
Interrupts	$-6.4723 \cdot 10^{-3}$	$4.4762 \cdot 10^{-3}$
Context switches	$-1.851 \cdot 10^{-4}$	$1.939 \cdot 10^{-3}$

Table F.8.: *Estimated coefficients β of linear regression model based on XMLValidate and SSJ*

Acronyms

3GPP 3rd Generation Partnership Project 9

ANSI American National Standards Institute 33

API Application Programming Interface 14, 31

APIC Advanced Programmable Interrupt Controller 37, 70

ASCII American Standard Code for Information Interchange 30

ASM x86-64-Assembler 36, 37, 54, 55, 58, 59, 60, 61, 62, 63, 64, 65, 66

BSS Business Support System 8

CDN Content Delivery Network 7

CFI Call Frame Information 36

CPI Call per Instruction 18

CPU Central Processing Unit 13, 14, 15, 17, 18, 19, 20, 21, 34, 37, 49, 50, 51, 53, 54, 64, 68, 70, 72, 73, 76, 78, 86, 90, 103, 104

CV Coefficient of Variation 72, 73, 75, 76, 77, 78, 79, 81, 83, 86, 90, 101, 102

DPI Deep Packet Inspection 1, 3, 6, 8, 9, 20, 29, 33, 49, 50, 51, 78, 90, 99

EPA United States Environmental Protection Agency 9

ETSI European Telecommunication Standardisation Group 7

GSL GNU Scientific Library 45, 52

GSO Generic Segmentation Offload 51

IETF Internet Engineering Task Force 9

IMS IP Multimedia Subsystem 7

IP Internet Protocol 12, 13, 49

IPv4 Internet Protocol Version 4 32

JNI Java Native Interface 39, 42, 46

JSON JavaScript Object Notation 30, 31, 32

JVM Java Virtual Machine 13

LRU Least Recently Used 23

MIP Mixed Integer Program 17

MSE Mean Squared Error 25, 45

MSR Model Specific Register 14

MTRR Memory Type Range Register 33, 34

MTU Maximum Transmission Unit 30

NAT Network Adress Translation 8, 49

NFV Network Functions Virtualization 1, 3, 5, 6, 7, 8, 9, 17, 19, 20, 26, 33, 53, 71, 72, 78, 83, 85, 86, 89, 90, 124

NFVI Network Functions Virtualization Infrastructure 8

NIC Network Interface Card 8, 19, 50, 51, 78, 79, 81, 104

NRDC National Resources Defense Council 5

OLTP Online Transaction Processing 15, 20

OS Operating system 13, 20, 35, 37, 53, 54

OSS Operation Support System 8

PAT Page Attribute Table 33, 34

PC Pin Control 14

PET Performance Event Trigger Framework 1, 3, 39, 40, 42, 43, 44, 45, 46, 47, 53, 71, 72, 76, 83, 85, 89, 90, 91, 99

PM Performance Monitoring 14

PMU Performance Monitoring Unit 14, 39

QoS Quality of Service 17

QPI QuickPath Interconnect 21

SDN Software Defined Networking 1, 3, 5, 8, 17, 89

SERT Server Efficiency Rating Tool 1, 3, 5, 6, 7, 9, 10, 12, 14, 15, 17, 73, 89, 90

SIMD Single Instruction Multiple Data 36, 54, 55, 58, 59, 60, 61, 62, 63, 64, 65, 66

SMT Simultaneous Multithreading 17, 59

SPEC Standard Performance Evaluation Corporation 1, 3, 5, 6, 9, 10, 13, 49, 51, 89

SUT System Under Test 3, 8, 10, 12, 13, 14, 15, 19, 20, 21, 29, 32, 33, 36, 37, 45, 46, 49, 50, 51, 53, 54, 64, 68, 83, 89

TCP Transmission Control Protocol 12, 13, 49, 51

TLB Translation Lookaside Buffer 18

TPC Transaction Processing Performance Council 9

UC Strong Uncachable 33, 34

UC- Uncachable- 33, 34, 36, 54, 55, 99, 100, 105

UDP User Datagram Protocol 20, 29, 33, 50

vCDN Virtualized Content Delivery Network 7

VM Virtual Machine 8, 17

VNF Virtual Network Function 5, 6, 7, 8, 9, 17, 20, 29, 49, 90

VNFaaS Virtual Network Function as a Service 7

VNFC Virtual Network Function Component 9

VNFD Virtual Network Function Descriptor 9

VNFI Virtual Network Function Instance 8, 9

VNPaaS Virtual Network Platform as a Service 7

WB Write Back 34, 64

WC Write Combining 33, 34

WDK Worklet Development Kit 12

WP Write Protected 34

WT Write Through 34

XML Extensible Markup Language 15, 20