# Design and Implementation of a Power-Aware System Exploiting Heterogeneous GPU–CPU Clusters

Bachelorarbeit im Fach Informatik

vorgelegt von

**Adam Wagenhäuser**

angefertigt am

**Lehrstuhl für Informatik 4**
**Verteilte Systeme und Betriebssysteme**

**Department Informatik**
**Friedrich-Alexander-Universität Erlangen-Nürnberg**

| | |
|---|---|
| Betreuer: | **Dipl.-Inf. Christopher Eibel**<br>**Dr.-Ing. Timo Hönig** |
| Betreuender Hochschullehrer: | **Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat** |
| Beginn der Arbeit: | **22. Dezember 2017** |
| Abgabe der Arbeit: | **22. Mai 2018** |

## Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

## Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Adam Wagenhäuser)
Erlangen, 22. Juni 2018

# ABSTRACT

Computing has many applications inducing the deployment of numerous large computer systems, which require significant amounts of energy, affecting the sustainability of the system and the supplying power grid. Computer system exhibit sudden changes in power draw, which threaten the power-grid stability, and the high energy demand requires steady cooling, together causing great costs. Therefore, a system is presented that obeys power limits, which increases the grid stability, improves the energy efficiency, which reduces the energy demand and thus the base cost. It also considers system activity in accordance with electricity prices, which cuts operating costs. The system achieves this by exploiting heterogeneity in computer systems and purposefully delaying pending activities. Utilizing heterogeneous processors improves the energy efficiency significantly, while it allows to adapt to varying power limits. Employing activity delaying allows to shift active times from periods of expensive electricity towards cheaper periods. Heterogeneous computer systems and flexible arrangement of activities, allow more ecological and economical operation of computing infrastructure.

# KURZFASSUNG

Die Datenverarbeitung hat viele Anwendungsgebiete was die Installation von vielen großen Rechnersystem zur Folge hat, diese haben einen signifikanten Energiebedarf, welcher wiederum Einfluss auf die Zukunftsfähigkeit der Anlagen und das versorgende Stromnetz hat. Rechnersystem zeigen abrupte Änderungen in ihrer Stromaufnahme, welche die Stabilität des Stromnetzes gefährden und der hohe Energiebedarf erfordert beständige Kühlung, was zusammen große Kosten verursacht. Daher wird ein System präsentiert, das Leistungsgrenzen einhält, was die Netzstabilität erhöht, die Energieeffizienz verbessert, was den Energiebedarf und damit die Grundkosten reduziert, welches ferner den Systembetrieb in Einklang mit Stromkosten betrachtet, was die Betriebskosten vermindert. Das System erreicht dies durch Ausnutzen von Heterogenität in Rechnersystemen und zielbewusstes Verzögern von ausstehenden Aktivitäten. Das Ausnutzen von heterogenen Prozessoren verbessert deutlich die Energieeffizienz und gleichzeitige erlaubt es sich an schwankenden Leistungsgrenzen anzupassen. Das Einsetzen von Aktivitätsverzögerungen erlaubt das Verlagern der Betriebszeit von Zeiträumen mit hohen Stromkosten hinzu günstigeren Zeiten. Heterogene Rechnersysteme und elastisch Anordnung von Tätigkeit erlauben ökologischeren und wirtschaftlicheren Betrieb von Datenverarbeitungsinfrastrukt.

# CONTENTS

**Abstract** **v**

**Kurzfassung** **vii**

**1 Introduction** **1**
1.1 Context . . . 1
1.2 Problem Statement . . . 1
1.3 Strategy . . . 2
1.4 Overview . . . 2
1.5 Publication . . . 3

**2 Background** **5**
2.1 High Performance Computing . . . 5
2.1.1 Workload Semantic . . . 5
2.1.2 Workload Managers . . . 5
2.1.3 Benchmark . . . 6
2.2 Programming Paradigms . . . 6
2.2.1 OpenMP . . . 6
2.2.2 OpenCL . . . 6
2.2.3 CUDA . . . 7
2.3 Energy . . . 7
2.3.1 Energy Efficiency . . . 7
2.3.2 Power Grid . . . 7
2.3.3 Energy Management . . . 8
2.3.4 Power Limit . . . 9
2.4 General Terms . . . 9
2.4.1 Heterogenety . . . 9
2.4.2 Client–Server Model . . . 9
2.4.3 Weak Symbol . . . 9
2.4.4 Quality of Service . . . 9
2.5 Related Work . . . 10

**3 Design & Implementation** **13**
3.1 Limo . . . 13
3.1.1 SLURM Workload Manager . . . 14
3.1.2 The Limo Components . . . 14

3.1.3 Power-Aware Job Distribution . . . 15
3.2 Profile Generation . . . 18
3.2.1 Architecture . . . 19
3.2.2 Protocol . . . 20
3.2.3 Client . . . 22
3.2.4 Server . . . 25
3.3 Cluster Manager . . . 27
3.3.1 Architecture . . . 28
3.3.2 Inter-Node Communication Protocol . . . 28
3.3.3 Workload Manager Algorithm . . . 32
3.4 Summary . . . 35

**4 Evaluation** **37**
4.1 Hardware and Sofware Setup . . . 37
4.1.1 Hardware . . . 37
4.1.2 Software . . . 39
4.2 GPU Frequencies . . . 41
4.3 Interactive Workload Characteristics . . . 44
4.4 Workload Management Limo vs. SLURM . . . 52
4.5 Summary . . . 55

**5 Conclusion** **57**

**Lists** **59**
List of Acronyms . . . 59
List of Figures . . . 61
List of Tables . . . 63
List of Listings . . . 65
List of Grammars . . . 67
List of Algorithms . . . 69
Bibliography . . . 71

# 1 INTRODUCTION

This chapter gives the context and motivation for this thesis, introduces the problems elaborated by this thesis, and the strategy and approach used to solve the stated problems.

## 1.1 Context

Information technologies are used in a wide range of applications (e.g., cloud computing, Internet of things, machine learning). With the wider spread of the computing sector the electrical-energy demand increases, too [Qia16]. However, a significant amount of energy demand is caused by the provisioning of increased compute capacity, only utilized during times of peak loads [Bar05].

The higher energy demand of computing systems causes higher thermal dissipation power, which in turn requires stronger cooling facilities for the computing systems, and becomes an increasing concern for system operators. Increasing the energy efficiency of computing systems became an active filed of research, as one solution in order to economically sustain the continuing growth of information-technology sector [BH07].

Another aspect of this thesis is motivated by the increasing deployment of renewable energy sources (e.g., wind turbines and solar panels) into the power grid. This trend poses new challenges for the grid operators, which have to maintain the grid stability (i.e., the equality between power generation and demand), caused by the volatility of the renewable energy generation, originating from natural fluctuations [IA09]. These fluctuations induce strong variations of the electricity price depending on the availability of renewable power in the grid [Ree17].

## 1.2 Problem Statement

This thesis elaborates three power-related concerns for operators of current and future large-scale compute clusters, in order to propose a solution for these: Power-grid stability, energy-demand reduction, and electricity-price considerations.

First, large over-provisioned server clusters (i.e., servers dimensioned for peak load) are most of the time utilized between 10 % and 50 % of their available compute capacity [BH07]. Consequently, at times of peak load the utilization levels rise significantly, causing rapid changes in the cluster's power draw. The power grid has to adapted the level of power generation to these changes in power draw. Since this adaptability is limited, computer systems have to limit their power draw variations, too, in order to ensure the stability of the power grid and avoid its collapse.

Second, higher energy efficiency reduces the energy demand of systems. This became a major concern in the sustainability for future compute systems like in exascale computing [Hem10].

Reduced energy demand also affects positively commercial concerns regarding the operation of such systems [Har+09]. Finally, the increased share of renewable power source, which are subject to volatile availability, in the power grid have an increased impact on the electricity price, which exhibits strong price fluctuations in accordance to variations of renewable power output [WS15]. Computer system operators have to consider these price fluctuations, in order to ensure cost efficiency.

## 1.3 Strategy

This paper introduces the workload manager Limo[1], which is designed to solve the problems given in the previous section. The following three goals are defined in order to solve these problems:

(1) Operate within a given power limit. A power grid operator can express the manageable scope of power draw variations, with an upper and a lower power limit. Consequently, Limo contributes to the grid stability by obeying these limits.

(2) Improve the energy efficiency of the cluster operation. This includes especially the reduction of the energy demand for the executed workloads.

(3) Monitor the current electricity price and reduce operational electricity cost by preferring workload execution during times of lower electricity prices.

The proposed workload manager implements a strategy based on the extensive incorporation of heterogeneous execution hardware, the employment of power caps on the hardware, and the shifting of load peaks by selectively delayed execution.

Heterogeneous hardware components within a system can significantly increase the energy efficiency, and thus being already deployed in a variety of systems, for example all of the 19 first entries in the Green500 list from November 2017, which rearranges the TOP500 list of the worlds most powerful computer systems by energy efficiency, contain heterogeneity through an additional accelerator processor such as an *graphics processing unit* (GPU) [Wan+17; Pro17a]. Beside increasing the energy efficiency, heterogeneous hardware components have different power-draw levels, which allows to obey a given power limit by utilizing the component that exhibits a power draw that lies within the power limit.

Enforcing a software power cap on a processor increases the power efficiency further [Wan+17]. Selectively delaying the execution of jobs allows, on one hand, to further comply with power limits since job with can not be executed at the given time of submit due to spent power contingent by other job executions, might be possible at a later point in time. On the other hand, job-execution delaying can be used to migrate execution from times of high electricity prices to times with lower electricity prices.

## 1.4 Overview

This thesis is split into five chapters. The first and current presents the introduction of the thesis, the second explains the background knowledge for the thesis, the third introduces the Limo concept and describes the two implementations, the fourth presents the evaluation results of three conducted experiments, and the last chapter draws an conclusion of this thesis.

[1] Limo is a German short term for lemonade, drawing an analogy to the workload manager SLURM, because its name is a lemonade like beverage in fictional world of Futurama [YJG03]. Limo is also the Esperanto word for limit, which highlights its awareness for power limits opposing to SLURM.

## 1.5 Publication

This thesis contains research results of the following paper, which have been published in a peer-reviewed workshop:

[Hön+18] Timo Hönig, Christopher Eibel, Adam Wagenhäuser, Maximilian Wagner, and Wolfgang Schröder-Preikschat. "How to make profit: Exploiting fluctuating electricity prices with Albatross, a runtime system for heterogeneous HPC clusters." In: *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '18)*. ACM. Tempe, AZ, USA, 2018, pp. 1–9.

In [Hön+18], I contributed the implementation of one prototype and its evaluation for the paper.

# 2 BACKGROUND

This chapter provides the background knowledge such as terminology and introduces the conventional concepts, which are the base of this thesis. Followed by the discussion of related work.

## 2.1 High Performance Computing

*High performance computing* (HPC) means any sort of computing which involves higher execution power then a single machine can provide. Therefore, HPC involves computing on a number of physical machines. This compound is called a *cluster*, and each machine a *node*. The following introduces terminology and

### 2.1.1 Workload Semantic

Many different domains make use computations (e.g., cryptography, physics simulation, web services). And those different applications have different characteristics and requirements. For instance, physics simulations are rather compute intensive and long running tasks. Whereas, web services are on average less demanding but require fast responses. Therefore, essentially two operation modes based on the application can be differentiated:

(1) *Batch operation*, where each task takes quite long, but are more flexible regarding the duration until the job finished.

(2) *Interactive operation*, where each task needs less time and resources, but responses are expected nearly immediately.

The batch operation might be considered the prevalent operation mode in HPC, since it is suited for heavier jobs, while the interactive operation prevalent in web oriented applications.

### 2.1.2 Workload Managers

Because HPC requires a lot of compute power one single computer is not sufficient for larger applications. Therefore, multiple machines are employed in order to rise the compute power. Each machine is called a *compute node* or just *node*. While all machines together are referred to as a *cluster*. The cluster is supposed to work as if it were one single powerful machine. So all nodes need to work somehow together. This collaboration in turn requires organization of the nodes in the cluster. One approach is to instate one machine—a *master* node, which take the task of managing the others (e.g., by distributing incoming work among the compute nodes).

The software performing the management of a cluster is called a *workload manager*. It is a distributed system running on all nodes, while usually the the master node takes a special role in the system which makes it a centralized distributed system. Formally, the workload manager is a program which takes as input a executable (a *job*) which is then executed by the system and finally returns the result of it (e.g., the standard output stream). The core feature of such a system is to distribute with constraints (e.g., number of compute nodes, amount of memory, hardware type) the given jobs among the available compute nodes. However, if the constraints can be fulfilled, there might remain a lot of options how, when, where to execute the job. These options allow for optimization of some aspect (e.g., energy efficiency, execution duration).

### 2.1.3 Benchmark

A benchmark is a reference point for comparing elements. In computing a benchmark is usually a program, the elements are different computer machines, and the values being compared are for instance the execution time or the energy consumed during that program execution. One important point to notice about evaluating benchmarks in computing is, that they commonly measure non-functional characteristics. It is generally expected that a benchmark would always produce the same functional result, however regarding floating-point arithmetic the results differ between architectures because their precision varies [HL04]. Furthermore, computer systems are today complex systems, designed to maintain only functional aspects. Therefore, non-functional aspects are subject to very high variations caused by a variety of sub systems and events. These variations are called *system noise*, because it appears as a noise in the evaluation data caused by the operating system [Tsa+05].

## 2.2 Programming Paradigms

In HPC applications a variety of programming languages and addition to existing programming languages have been developed. One might say that one of the earlier HPC languages is Fortran, as is a language developed to be used for solving numeric problems, which still occupy a large use-case for HPC systems. Despite, Fortran in used until today, it no longer widely used. The C programming language is neither developed for solving numeric problems nor for parallel computing, needed for todays multi core and many core processors, which also make up HPC clusters. Nevertheless, C become a quite popular language and is used in many domains. In order to exploit that popularity, several additions have been developed for the C language.

### 2.2.1 OpenMP

One such extension of the C programming language is *open multi-processing* (OpenMP) from the OpenMP Architecture Review Board. OpenMP is a paradigm which requires support of the C compiler. But is therefore platform independent like C itself, and even available for accelerators. OpenMP is intended to simplify the development of parallel applications [Ope13].

### 2.2.2 OpenCL

*Open computing language* (OpenCL) instead is a C compatible *application programming interface* (API) from the Khronos Group. This means, that no special compiler support like for OpenMP is required. But instead a special run-time environment on the target system needs to be available, which is an OpenCL loader and at least one OpenCL driver for an OpenCL capable device [Khr15].

An OpenCL application is split into a host code (e.g., normal *central processing unit* (CPU) binary utilizing the OpenCL API) and device code or *kernel* (i.e., the part of the application than is executed on the OpenCL device). One specialty of OpenCL is that kernel source code must be present for the application at run-time, because it is just-it-time compiled, which allows OpenCL to be platform independent [Khr15].

### 2.2.3 CUDA

*Compute unified device architecture* (CUDA) from the Nvidia Corporation is very similar to OpenCL and bases on similar programming paradigm. CUDA applications are also split into host and device code, however, the device code is compiled in advance. This in turn is possible because only the GPUs from Nvidia are supported, which is ensured by Nvidia because the CUDA API is proprietary standard unlike the OpenMP and OpenCL [Nvi18].

However, it exist claims that applications based on CUDA run faster and more efficient on Nvidia GPUs than the an equivalent application based on OpenCL, and the results presented in this thesis also show that the CUDA implementations are mostly more energy efficient than the respective OpenCL implementation.

## 2.3 Energy

This section introduces terminology related to energy and power.

### 2.3.1 Energy Efficiency

Efficiency is the avoidance of overhead. Usually used in a graduated sense, and thus the reduction of overhead. Energy efficiency means therefore the reduction of dissipation power. Since computer systems run on electrical power, and thus having an energy demand, energy efficiency applies to them. However, there are plenty of points introducing overhead.

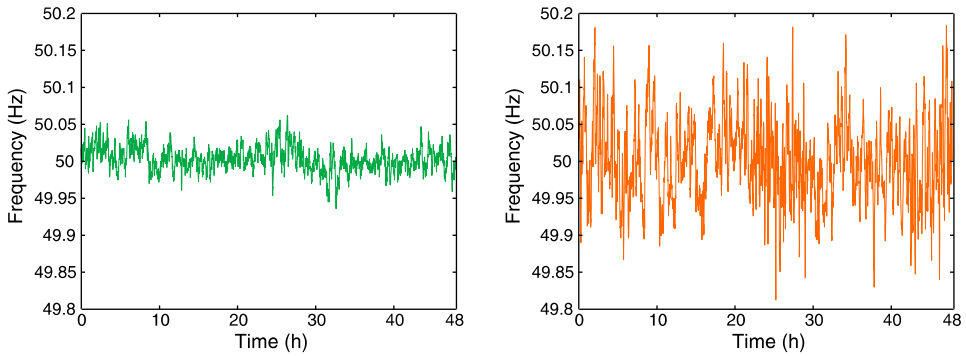
For example, assume a mathematical calculation, which here is the purpose to be carried out. Since mathematical calculation do not inherently require any energy, all energy used during the execution of the calculation poses overhead. The energy needed is the electrical energy dissipated by the hardware machine on which the calculation is carried out. First of all the power supply unit and every conductor within the machine have a efficiency factor (i.e., a metric of efficiency where 1 is prefect efficiency and 0 means only overhead) less than 1. Further the CPU requires specific amount of energy for each instruction carried out. Next, the OS adds additional instructions in order to carry out is purpose, which however, is overhead regarding the actual task. Last, the implementation of the mathematical algorithm might contain unnecessary instructions adding further overhead.

One application of energy efficiency is to be used in HPC benchmarks, to compare HPC systems by energy efficiency. In order to compare them, a metric describing energy efficiency is required. A simple metric is to compare the energy demand of the system required to process the benchmark. When confronted with multiple benchmarks other metrics are used, which are independent from the benchmarks such as operations per energy (i.e., Op/J) metric [Har+09].

### 2.3.2 Power Grid

The power grid the infrastructure that connects the electrical power produces (i.e., power plants) with power consumers (e.g., computation center). The operator of the power grid is responsi-

ble for ensuring the grid stability, by balancing the amount of produced and consumed power. Strong variations in voltage or utility frequency indicate weaken grid stability (see Figure 2.1). An emerging challenge for the grid operators pose the rising incorporation of renewable energy sources (e.g., photovoltaics and wind turbines), because there varying availability threaten the grid stability [IA09].




**(a)** Central Europe **(b)** Great Britain

**Figure 2.1** – Variation of utility frequency over 48 hours for two european countries [Wik11]. Shows differences in the grid stability.

### 2.3.3 Energy Management

Energy or power management is a hardware feature of electronic systems such as computer. It is intended to reduce the power draw and/or energy demand of the system. This is achieved by switching off unused subsystems, entering sleep states (i.e., reducing the activity by slowing the clock) such as P-states during idle periods. Computer systems provide power management via the *advanced configuration and power interface* (ACPI) and *advanced power management* (APM) standards. One feature and API that allows fine grain power management for a software application is Intels *running average power limit* (RAPL) on x86 CPUs [Int18]. RAPL allows especially to specify a power cap, which effectively limits the power draw of the CPU. The presented features are all CPU related, however, GPU manufacturer also implemented power management features. One GPU-related power-management tool is *nvidia-smi* for Nvidia GPUs [Nvi16]. It support voltage and frequency scaling as well as power caps. However, the Nvidia Quadro P2000 used is the evaluation of this thesis did not implement all the power-management features supported by nvidia-smi, and only frequency scaling is available.

Further concepts in the domain of energy or power management are *dynamic voltage scaling* (DVS), *dynamic frequency scaling* (DFS), and the combined *dynamic voltage and frequency scaling* (DVFS). These concept propose to variate the voltage and frequency of a processor at run-time, in order to adapt the processor performance to the current needs of running application. Consequently, these concept are implemented in software at OS level (called power governor under Linux), or at application level. The implementations using feature offered by ACPI, APM and RAPL.

### 2.3.4 Power Limit

In contrast to a power cap from the previous section, a power limit is constraint for subject system. It is specified as a power value $L$. The subject system is required to have a power draw $P$, which is below the specified value such as $P \leq L$.

## 2.4 General Terms

This section introduces further terminology and concepts.

### 2.4.1 Heterogenety

Heterogeneity in computer systems is a rising phenomenon. An increasing amount of systems incorporate heterogeneity for various reasons (e.g., to conserve energy or to optimize particular tasks). There are essentially two sorts of heterogeneity: 1) systems consisting of nodes with the same architecture but different compute power such as ARM big.LITTLE [ARM13], 2) systems consisting of nodes with different architectures (e.g., CPU and GPU). There are different scopes of heterogeneity: 1) system scope, 2) component scope. 3) configuration scope.

### 2.4.2 Client–Server Model

Take two machine and put a conductor between them, call the first *server* and the second *client*. Now the client issues a request via the conductor and the server replies. We shall call this the client–server model. A common protocol used today is *transmission control protocol* (TCP) which is placed in the transport layer of the *international organization for standardization* (ISO) *open systems interconnection* (OSI) network model and hence usable for applications. The other protocols used in that network stack involves Ethernet, *internet protocol* (IP) and said TCP.

### 2.4.3 Weak Symbol

Weak symbols in programming are a way of declaring optional symbols within a program. Programs and object files (i.e., a compiled C module) such as *executable and linkable format* (ELF) files contain symbols. Symbols similar to functions and variables in C are either declared (e.g., an extern global variable in C) or defined (e.g., a function with body). If defined, a symbol contains an arbitrary value such as the memory address of a variable, or if declared, the symbol is required from another object file, but might be used as if locally defined.

In order to be executable, all declared symbols of program must have exactly one definition. However, weak symbols have here a different semantic: weakly defined symbols might be overridden by a strong symbol (any non-weak symbols), and a weakly declared symbol does not need to have any definition, and defaults to zero. The latter can be used to declare an optional feature (e.g., a function) and test at runtime if the address of the feature is zero (i.e., not present) or not (i.e., available).

### 2.4.4 Quality of Service

Quality of service is the ‘totality of characteristics of a telecommunications service that bear on its ability to satisfy stated and implied needs of the user of the service’, as defined by the ITU [ITU08]. This means that QoS is the accumulation of service characteristics which has significance to the user

(e.g., duration until the end of execution). This thesis will especially consider QoS such as energy demand of the execution and power draw exhibited during the execution of an job.

A user of a service with QoS would state (explicitly or implicitly) levels of quality (e.g., an maximum amount of energy which might be used for the execution and the latest point in time the execution should finish). Such a statement is called QoS *requirements of user*.

## 2.5 Related Work

This section presets the research conducted prior to this thesis and discuses the differences and improvement of this thesis over the previously conducted work.

Lin et. al. presented user-driven frequency scaling (UDFS), which is driven by the user feedback, and process-driven voltage scaling (PDVS), which is driven by recorded optimal voltages for CPU given frequency and temperature. This work requires direct manual user input, which is assumed to be a burden for the user. Instead, this thesis proposes an automated system which requires no manual input at run-time [Lin+09].

Meisner et. al. proposed the approach PowerNap, which reduce the energy demand of servers by switching them off, while no clients are using the respective server. This approach can only exploit true idleness of a single machine. In contrast, the approach of this thesis utilizes gradual methods (e.g., Intel's RAPL), which allows to reduce energy demand at arbitrary level of machine utilization, even during full load [MGW09].

Mei et. al. conducted an elaborated impact analysis of GPU voltage and frequency scaling on the application execution. This analysis is very similar to the first experiment of this thesis. However, there are discrepancies in the results. Notable, Mei et. al. report impact of memory frequency scaling, which on the contrary, was not observed in the experiment of this thesis. An additional difference is the evaluated hardware, Mei et. al. evaluated a Nvidia GTX980 and GTX560Ti, whereas this thesis evaluates a Nvidia Quadro P2000. Finally, this thesis focuses on exploiting results instead of analyzing them in great detail [MWC17].

Wang et. al. presented a benchmark suit, which bases significantly on the Rodinia suit also evaluated in this thesis, and use it evaluate different processor including an ARM Cortex-A53, which is also evaluated in this thesis, an Intel Xeon E5-2630v3, which is similar to the Xeon evaluated in this thesis, and a Nvidia GTX980 like Mei et. al. did. Wang et. al. evaluated the energy demand for these system during execution of their benchmark suit. Additionally, they evaluated the impact of thread counts on the CPUs and frequency scaling on the GPUs. This paper researches similar characteristics as the first two experiments of this thesis, however, the same discrepancies regarding the GPUs as compared to the results of Mei et. al. are shown. Further, this thesis evaluates the impact of power caps on the Xeon CPU [Wan+17].

| Feature | Tivoli | Moab | Univa | SLURM | SLURM-HC | LIMO |
|---|---|---|---|---|---|---|
| Job pinning | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |
| CPU allocation | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ |
| Quality of service | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Generic resources | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Cluster status | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Heterogeneity aware | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ |
| Power/price aware | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |

**Table 2.1** – Comparison of different workload managers [Hön+18].

Table 2.1 table the feature set of Limo compared with four other workload managers. Trivoli shows the least feature set is not further discussed, in favor of the workload managers with richer feature set. Commercial workload managers such as Univa Grid Engine [Uni18] and Moab [Ada18] are already prepared for heterogeneity support, however, in contrast to Limo, they do not take power or energy demand into consideration.

The SLURM workload manager by Yoo et. al. served as inspiration for Limo, thus both workload manager share many features and concept, as well as the similarly named SLURM-HC by Wagner. Wagner extended the feature set of SLURM by heterogeneity awareness. Limo further add power and price awareness into the workload management [YJG03; Wag17].

In summary, the basis of this thesis are already covered in the scientific community and parts of the presented experiments were already conducted before. Though, the innovation of this thesis is the combination of these primary results and the further exploitation of them to obey power limits and to direct electricity-price aware operation.

# 3 DESIGN & IMPLEMENTATION

In this chapter, the concept and prototype implementations of Limo are described. The first section starts to outline how workload managers work in general and continues with the approach of Limo, which considers energy efficiency aspects, power limits and price awareness. In order to achieve this, Limo requires additional information. One way to provide them is to generate them in advance. Section 3.2 describes a prototype implementation of a generator for profile data in the domain of interactive applications. The last section (Section 3.3) puts the pieces of this chapter together. It presents the prototype implementation of an HPC-suitable workload manager for batch applications, which concentrates on power awareness and price awareness.

## 3.1 Limo

Workload management is the distribution of jobs over a compute-node cluster in accordance with a set of constraints (i.e., QoS requirements of user) (see Section 2.4.4). As explained in Section 2.1.2, after taking the constraints into account, the workload manager still has some degrees of freedom for distributing a job. This freedom allows to select a node regarding special criteria and thus optimizing the operation along that criteria. Contemporary workload managers already implement some optimization goals such as minimizing execution time through preferring the most powerful nodes first, or increasing load balancing by giving each node the same amount of work.

However, common workload managers allow only to select constraints out of a predefined set, which rarely includes a power limit (see Section 2.3.4. Similarly, workload managers which allow to optimize operation towards better energy efficiency are quite rare, as well as the awareness for heterogeneity (see Section 2.4.1) within the cluster. Therefore, the concept of the workload manager Limo is introduced, which is well aware of the cluster heterogeneity, and exploits it in order to improve energy efficiency in accordance to a given global power limit (see Section 2.3.4).

The combination of heterogeneity, energy and power awareness is quite reasonable. First, heterogeneity in computer systems is a rising phenomenon. An increasing amount of systems incorporate heterogeneity for various reasons (e.g., to conserve energy or to optimize particular tasks). There are essentially two sorts of heterogeneity: 1) systems consisting of nodes with the same architecture but different compute power (e.g., ARM big.LITTLE [ARM13]), 2) systems consisting of nodes with different architectures (e.g., CPU and GPU). Second, such heterogeneity is a notably good base for improving energy efficiency and maintaining a power limit. This is because as shown in Section 4.3, the same task has different execution times and different power draws on diverse hardware, which means it consumes different amounts of energy. Also the same hardware is variously utilized by diverse tasks, which causes the hardware to draw varying amounts of power.

Knowing the impact of heterogeneous compute nodes, Limo optimizes energy efficiency by selecting the best compute node for each given job. This requires detailed information about the characteristics of the expected jobs, hardware, and the results of running the one on the other. Therefore, Limo maintains a profile database about the energy demand and time consumption of previous job executions.

### 3.1.1 SLURM Workload Manager

Limo is a specialized workload manager, which concentrates on energy aspects (e.g., obey power limits, improve energy efficiency). Section 2.5 presents several contemporary general-purpose workload managers. SLURM is one of them which has a quite rich feature set and is *open-source software* (OSS), which makes it ease to work with, and hence, SLURM is chosen as base for Limo, which essentially extends its feature set. In order to introduce to the fundamentals of Limo's concept, the core components of SLURM are presented.

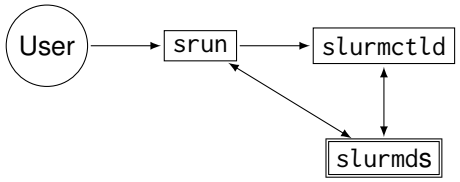



**Figure 3.1** – Overview of SLURM's components (based on [YJG03]).

SLURM is composed of three core components, which are named after the executable implementing that component. Figure 3.1 shows these components: the job submission interface (`srun`), the management unit (`slurmctld`), and the executors (multiple instances of `slurmd`). Deployed as a workload manager of a cluster as introduced in Section 2.1.2, `slurmctld` runs on the master node, while `slurmd` runs on each compute node. `srun` is either executed on the workstation of the user or, depending on the deployment, when the user has accesses directly the master node, it is executed there.

The `srun` command connects the user with the `slurmctld` process, and manages input and output forwarding to and from the user. Unlike the other executables, `srun` is an ad-hoc command, which is just running when the user has currently a job in execution.

On the contrary, `slurmctld` is a daemon processes, which is active all the time and waits for the submission of jobs. Whenever the user submits a new job with `srun`, it sends `slurmctld` the specification of the job (e.g., the executable and the constraints regarding its execution). `slurmctld` then searches for an available compute node which fulfills the given constraints (e.g., hardware features). If there is no such node, SLURM delays the job until an appropriate node becomes available. If there are multiple matching nodes, SLURM takes just one of them. And, the job is transmitted to the `slurmd` of the selected node.

The `slurmd` daemon of a compute node waits until it receives a job. When it gets one, `slurmd` and `srun` establish a data connection between each other for input transmission. Then, `slurmd` executes the requested job and sends the result back to the originating `srun`.

### 3.1.2 The Limo Components

As earlier discussed, SLURM ignores heterogeneity of the compute nodes. Hence, if SLURM is used with a heterogeneous cluster (e.g., with nodes of divers compute power), it could lead to

quite poor job distribution resulting in non-optimal load balance within the cluster (see [Wag17]). Consequently, SLURM also misses to exploit such heterogeneity to improve energy efficiency and to maintain a power limit.

Therefore, Limo extends SLURM by three more components which provide the further information used to improve the job distribution algorithm. First, an energy meter measures the present and actual power draw of the cluster and provides that information to the master component. Second, an energy spot-market–price monitor provides the current energy price. Third, a grid power-limit monitor provides the effective power limit for the cluster.

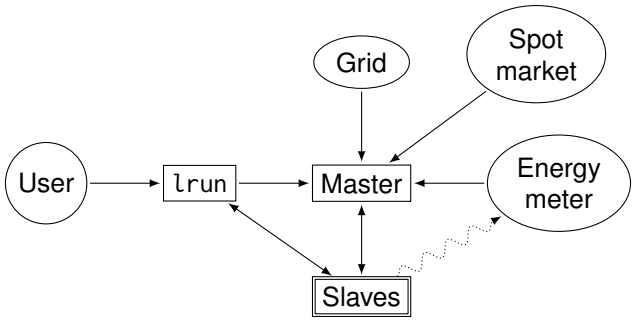



**Figure 3.2** – Overview of Limo's components.

Figure 3.2 shows the components of Limo. The `lrun`, *master*, and *slaves* elements correspond to the `srun`, `slurmctld`, and `slurmd` elements respectively in Figure 3.1 of SLURM's components. Since these executable where prefixed by an S for SLURM and are now prefixed with an L for Limo. In contrast, the *grid*, *spot market*, and *energy meter* elements represent the introduced components.

The grid and spot-market components are some sort of gateway components, which request information from some remote resource (e.g., via a web API) and present them to the `lmaster` in a convenient format. The energy meter is a different component, which requires some sort of additional hardware to perform the measurements, such as an external power meter or a special CPU which integrates a feature for power measurement such as Intel's RAPL feature of x86 CPUs, which enables the CPU to measure its own power draw.

### 3.1.3 Power-Aware Job Distribution

The major extension by Limo is not directly shown in Figure 3.2, since it is the improvement in the job distribution algorithm with the additional information provided by the added components by the implementation in `slurmctld` resulting in Limo's `lmaster` component. This additional information information is composed of the actual cluster power draw, the power limits, and the energy price.

Power limits have the effect of smoothing the cluster power draw curve, because narrow limits would not leave the space for too strong variations. This effect of fewer power draw variations is essentially good for the power grid because, rapid power draw variations threaten its stability. The power grid operator has the obligation to maintain grid stability, and consequently, the operator has to encourage his subscribers to minimize their variations (e.g., by providing power target and penalty zones). Computer systems, however, usually exhibit rapid power variations and large-scale clusters have a significant power draw, which impacts the grid stability. Limo takes such a power target zone as high and low power limit and tries to stay within those limits. Figure 3.3 visualizes this concept.

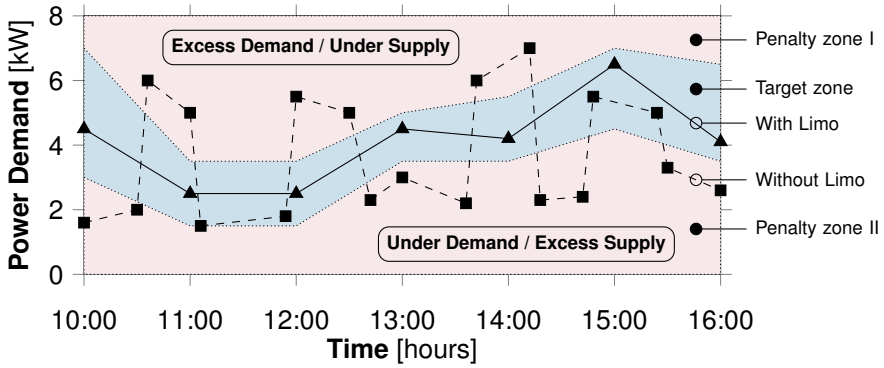



**Figure 3.3** – Dynamic power limits forming penalty zones [Hön+18].

Further, power grids (see Section 2.3.2) with a significant amount of renewable power sources are subject to high supply fluctuations, originating from power plants such as photovoltaics and wind turbines, which rely on the presence of sunshine and wind flow, respectively, which in turn are subject to natural fluctuations (see Section 2.3.2). Some power grids allow energy to be traded at a spot market, where supply fluctuations result in varying energy prices. Computer clusters with relaxed job deadlines can exploit these price variations. Hence, Limo shifts job execution towards times of lower energy cost by best effort.

Considering the energy price and power limits, Limo uses the following job distribution algorithm. Figure 3.4 outlines the algorithm: when Limo receives a new job, it looks for available compute nodes which fulfill the stated constraints of the job. If there is no such node, the job gets enqueued and the processes starts later (e.g., when a slave finished executing a job) again. If one or more possible nodes were found, then depending on the energy price either the node with the lowest energy demand or the node with the shortest execution time for the given job is selected. Then Limo tests if the power draw of the system together with the expected additional power draw would still conform to the present power limits. The expected power draw is estimated from the recorded power data of previous executions of that job on the selected node's hardware. If the power limits would stay satisfied, the job is forwarded to the compute node and executed. If the power limits would be violate, the job is enqueued and the process is tried again later.

The reason for the decision to either dispatch a job on the most energy-efficient compute node or the fastest compute node are as follows: On the one hand, when the energy price is high, saving energy and therefore cost is quiet reasonable. On the other hand, when the energy price is low, the fastest execution might be preferred to execute as many jobs as possible as long as energy is cheap.

**Node Configuration.** A further characteristic, especially regarding energy efficiency of a job–node relationship is how the power management facility (see Section 2.3.3) of a node is configured. This includes what OS process governor is in use and to what frequency and voltage the computation unit is tuned to by the mean of DVFS (e.g., with tools like RAPL) (see Section 2.3.3). As Section 4.2 shows, GPU DFS has an impact on the duration and power draw of the execution and as Section 4.3 shows, DVFS on a CPU has also an impact on the energy efficiency of a complex interactive workload.

**High vs Low Power Limit.** The job dispatch algorithm of Limo is quite strict about the power limits, as it is the last check to prevent any limit violation. Anyhow, the test should not be taken too strict, because if the system is currently below the lower power limit and a small job is about to be dispatched, it is better to dispatch it even if the system would still draw not enough power, and

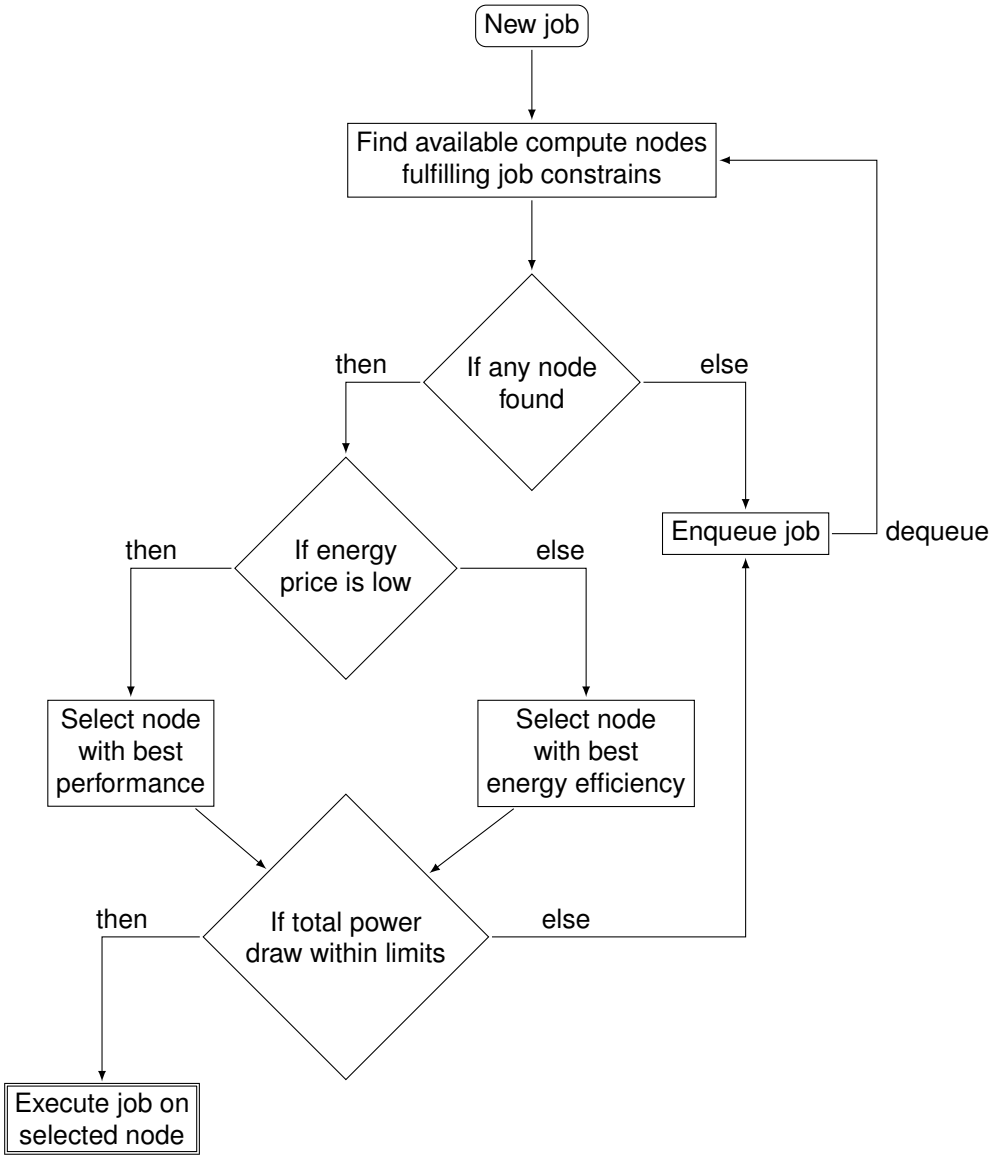



**Figure 3.4** – Scheme of Limo's dispatch algorithm.

thus not meeting this criteria. Therefore, the upper limit is the more important one, and the one which should be strictly enforced, unlike the lower limit. Nevertheless, the lower limit is an indicator for Limo to increase its power usage. In order to do so, Limo might rise some power management values (e.g., increasing a RAPL power limit), or start ad-hoc job if they could be run when needed (e.g., self-assessment tasks, crypto-currency mining).

Inspecting the power limits further reveals that Limo is just supposed to stay within those limits. This might be difficult enough assuming a narrow target zone. However, if the target zone is rather wide, it raises the question whether Limo should rather target for the bottom or the top of the zone. Just regarding the energy price, the answer is rather simple: target the top during times of low

energy prices and target the bottom while energy prices are high. On the other hand, it might be somewhat more complicated in practice, for instance when taking the amount of pending jobs into account. So if there are many queued jobs during times of high prices it might be still reasonable to dispatch so many jobs until reaching the high power limit, or if there are just a few jobs while prices are low, it might be the better choice to stay at the lower power limit, to avoid running out of executable jobs.

**Platform Independence.** Giving Limo access to multiple possibilities for dispatching a job helps to effectively employ Limo. Thus, it is assumed that jobs are platform independent, and can be executed on any node in the cluster regardless of the architecture of the node. This can be achieved in a number of ways. One way would be to just package the object code for the different architectures together into one binary—a so called *fat binary* [Gor15], this approach allows to directly execute native executable utilizing all special instructions directly, however requiring to compile the application explicitly for all available platforms and increasing the size of the executable as many times. An alternative way to archive platform independence is to use an unified virtual machine, which could executes a platform-neutral object code on any node [FK97]. The second approach introduces a just-in-time compilation step which increases run time and might not be as good in utilizing special hardware features but does not increase the executable size.

This section introduced a concept for a heterogeneous-aware cluster manager which incorporates power and price awareness, and hence is capable of exploiting varying energy prices while obeying given power limits. The following section present a prototype program for generating a profile database necessary for the design and the last section of this chapter present a prototype implementation of here presented concept.

## 3.2 Profile Generation

One crucial point for the correct operation of Limo is profile database present at run time for estimating the power draw of the job execution on compute nodes. While this database can be extended during operation, it needs first to be initialized. Therefore, this section presents the prototype implementation of a program, which executes a given job on a specific hardware and measures the power draw and duration of that execution, which can be used to create such database.

This prototype is specialized to examine interactive workloads (i.e., sets of many short-running jobs, see Section 2.1.1). To gather reliable data, the prototype needs to exhibit the characteristics of a benchmark as introduced in Section 2.1.3. The short run times pose a challenge for evaluating these jobs, because measurements would need to be precisely timed. A common technique to solve this challenge is to run a job multiple times in a row, which increases the time frame for the measurement, and the result is the average of these runs, which reduces the impact of system noise.

A speciality of interactive applications (i.e., web servers) is that the system load varies strongly depending on the number of jobs per time frame (e.g., as a result from the number of users), in contrast to a batch system, where a single job is usually sufficient to fully occupy the system for a significant amount of time. The impact of the system load on power draw and energy efficiency in the job processing is an aspect worth investigating; therefore, the prototype allows to examine the system at different levels of utilization by considering the number of jobs per time.

In the following, the term *workload* will be used to refer to one specific set of jobs with only one type, it is specified by the number, sequence, and input (e.g., a matrix for a matrix-decomposition job) of the jobs in the set. The prototype is designed to be independent from the type of job, hence

it is except for the implementation of the jobs' execution algorithms. The implementation of an execution algorithm for one job will be called an *application*.

Next, the general architecture of the prototype is introduced, followed by the presentation of the client–server communication protocol used (Section 3.2.2), and the details of the client part (Section 3.2.3) and server part (Section 3.2.4) of the prototype.

### 3.2.1 Architecture

The software architecture of the prototype bases on the client–server model (see Section 2.4.2), where the server is the executing part and the client provides the workload, which is entirely processed during one connection—subsequently called a *session*. The server handles only one client at a time, resulting in a one-to-one relationship between client and server, because handling a client means processing a workload and when handling multiple clients at a time the two processes would interfere and yield wrong results.

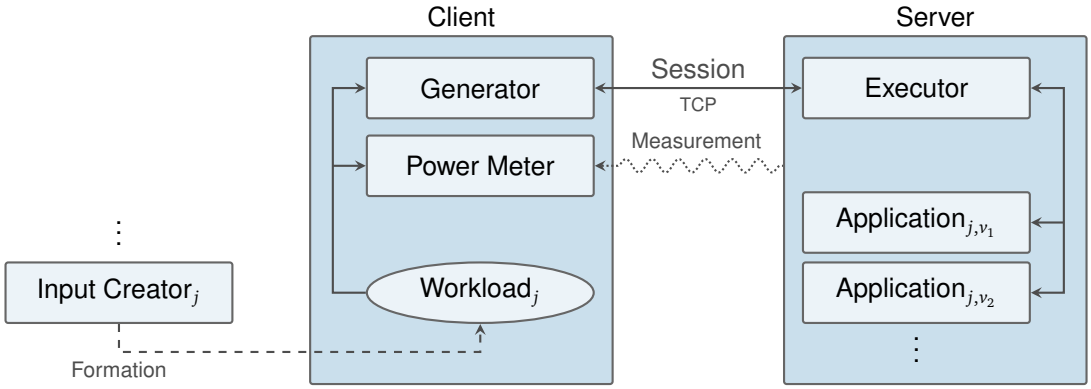



**Figure 3.5** – The components of the profile generator prototype.

The motivation for choosing the client–server model is that the system which is measured should only be occupied with the execution, and the network allows to outsource other tasks such as the power measurement and job provision. When separating execution and job provision, the jobs need to be transmitted across the boundary, for which TCP over an IP network is a convenient solution in the context of client–server architectures, because it is connection based and provides a reliable stream abstraction on top of IP.

Figure 3.5 shows an overview of the prototype architecture. Summarizing it, there are five functional units—the *modules*: executor, applications, generator, power meter, and input creators. They can be categorized by two characteristics: 1) server and client code—shown from the right to the left side respectively, and 2) job-specific and job-agnostic code—shown bottom to top, respectively.

The server part represented by the right blue box consists of the executor and applications. The client side represented by the middle blue box consists of the generator, power meter, and a workload. The input creators are at the left-hand side of the client. They are not part of the client scope (blue box), because that scope contains only the relevant components for the client to be executed where the input creators do not belong to, since they can be run in advance and only a created workload file is required.

Workload and input creators are indexed with $j$, because a workload file is specific to one type of job, and hence requires a proper input creators. The applications are also specific to a type of job

$j$ and additionally there might be multiple versions $(v_1, \ldots, v_n)$ for the same job type. The other modules executor, generator, and power meter are generic; that is, they work with any kind of job.

The prototype is mainly implemented in the C99 [ISO99] standard of the C programming language and the POSIX.1b API [IEE93]. Only the specific application execution modules are implemented using code written using the _GNU_SOURCE API [Lin17] and one of the three C-based dialects and APIs: OpenMP, OpenCL, and CUDA (see Section 2.2).

### 3.2.2 Protocol

The client and server communicate using TCP as transport. Most of the time the client sends jobs and the server executes them. However, this protocol is not unidirectional because the initialization and finalization phases require bidirectional communication. This subsection describes the communication protocol between the client side and the server side of this prototype, starting with the message types and followed by the message sequence required by this protocol.

Listing 3.1 shows the C structures representing the messages used in the communication protocol. The most frequent message type is the job-transmission message `StreamPackage`, which contains the input data for the job executions. The size in the `StreamPackage` is a signed integer. The reason for this is that `StreamPackage` and `StreamStatus` are actually a union; each transmission is distingushed by the sign of `size` or `code` value, respectively.

Transmissions with negative values are treated as a `StreamStatus`, thus without an application invocation. Transmissions containing a positive size value represents consequently `StreamPackages`, with a `content` consisting of as many bytes as `size` specifies. On special case is a `size` value of zero, which is just ignored, because it is assumed that the job processing needs a non-zero–sized input. The `sqn` value is a client-specific value, intended for logging and debugging purposes only.

```c
struct StreamPackage {
  uint32_t sqn;       // the sequence number of the package
  int64_t  size;      // the size of content in bytes
  char     content[]; // the application-specific task input
};

struct StreamStatus {
  uint32_t sqn;       // the sequence number of the package
  int64_t  code;      // the status code
};

#define STREAM_INIT_NAME_SIZE 0x80
struct StreamInit {
  uint64_t magic;     // magic number to guard against random connections
  uint64_t version;   // version number to guard against incompatible peers
  uint16_t app;       // the id of the test application to start
  char     name[STREAM_INIT_NAME_SIZE]; // a custom name for the run
};
```

**Listing 3.1** – Connection protocol data

The `StreamInit` structure is only once transmitted during a session. As the name suggests, it is used for sending initialization information from the client to the server. It first contains a `magic` and a `version` number which are intended to ensure compatibility between the server and the client. The magic number is constructed to detect endianness incompatibility between the participants, which is important because conveniently the host byte order is used in messages. The field `app` of

`StreamInit` is actually the most important value for the purpose of the protocol, it contains the *identifier* (ID) of the application module to be used, hence defining the type of the jobs which are expected to follow.

It is worth mentioning that there are multiple implementations for the same type of job, thus a job might be processed by different applications, whereas a single application is specific to its type of jobs. One notable exception is the dummy application, which can process any job, since it just discards them. Last, the structure contains a session `name`, which is intended for log-file matching since the server is supposed to process multiple sessions with possibly different clients. Eventually, it is used to convey additional application-specific configuration arguments.

**Table 3.1** – Protocol status codes

| Name | Description | Type |
|---|---|---|
| `OK` | To be ignored | Info |
| `ERROR` | Indicates general error | Error |
| `WRONG_VERSION` | Indicates incompatible peers | Error |
| `INVALID_REQUEST` | Indicates an application ID unknown by the server | Error |
| `AWAIT_START` | Signals that the server is ready for the new session | Control |
| `ACK_START` | Marks the server ready to process jobs | Control |
| `FINI` | Introduces the finalization of a session | Control |
| `ACK_END` | Marks the successful end of the session | Control |

Table 3.1 shows the different code values used in the `StreamStatus` message. They are subdivided into the three types info, error, and control. The info type consists only of the `OK` code, which is essentially a do-nothing operation or no-op. Conveniently, it is encoded as zero and as mentioned before, this means it represents also a zero-sized package, which in turn is equally ignored. The second class of status codes are the error codes: `ERROR` represents any arbitrary error state during the session, and `WRONG_VERSION` and `INVALID_REQUEST`, which both are initialization errors, are differentiated to give more user-friendly error messages. The final class of status codes are special control indicators for the session management. The usage of the control codes is shown in Figure 3.6 and explained in the following paragraphs.

The chronological sequence of messages is shown in Figure 3.6. The initiator of the protocol is the client or more specifically the generator, which starts by opening a connection to the executor module of the server. Since the executor is designed to process only one client at a time, the establishment of the connection could take as long as the processing of a preceding client's session would take. To minimize inference, the generator waits passively until the executor becomes ready and sends the `AWAIT_START` status code. Then, the client creates the `StreamInit` message and sends it to the server, which initializes itself for the new session and starts the requested application. When the initialization has finished, the server answers the client with the `ACK_START` status code.

Upon receiving the `ACK_START` status code, the client initiates the time measurement, starts reading the workload, and transmits the specified jobs with the specified timing (i.e., by waiting the stated delay before sending it). This process of reading, waiting, and sending is repeated until the end of the workload has been reached. One may notice that the executor is never responding to the job execution requests from the client. This is quite reasonable, because the purpose of this prototype is to measure the energy consumed while processing the workload, not individual jobs.

Furthermore, the client can send with its maximum capacity not required to wait for any response, if the server dose not acknowledge the processing of a job to the client. This way a single client can

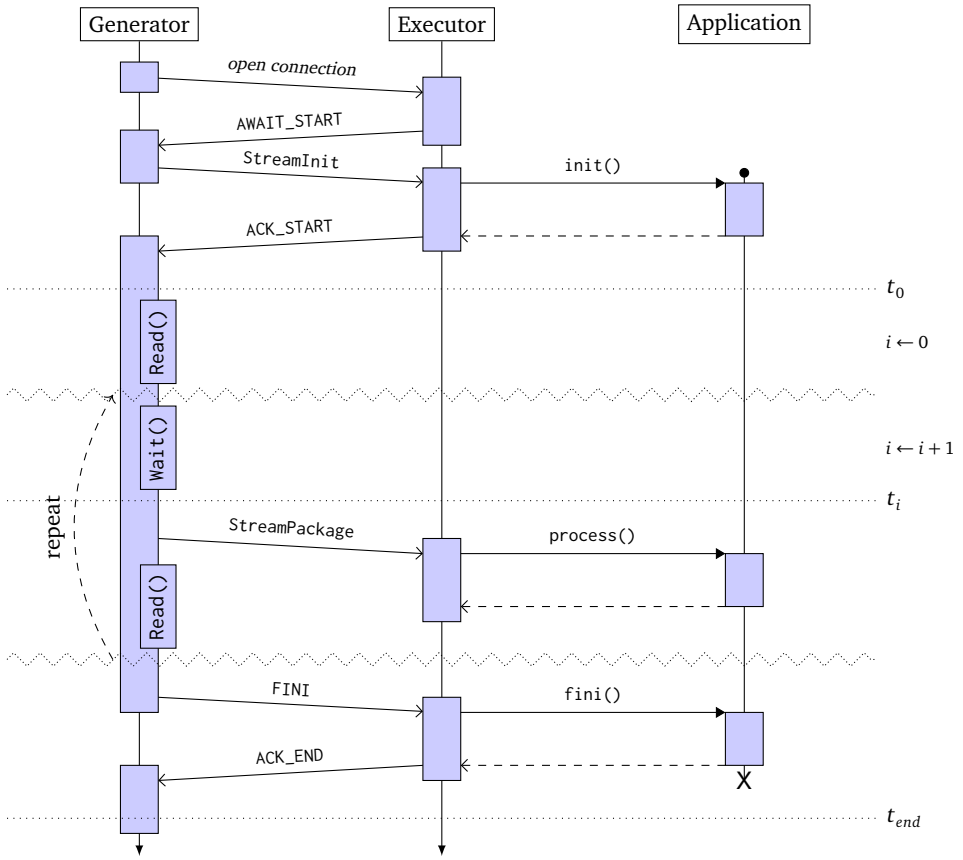



**Figure 3.6** – Profile generator communication protocol.

cause the full utilization of the server. However, if the sever can not keep up with the load from the client, TCP will employ its flow control mechanism slowing down the client transmission.

Finally, after transmitting the last job, the generator sends the `FINI` status code to the executor. After processing all buffered jobs the executor terminates the application and returns the `ACK_END` status code to the generator. The generator stops the time measurement on arrival of that status code, which concludes the session.

### 3.2.3 Client

The purpose of the client is to send a set of jobs to the executor in a comparable way, hence is should be a benchmark (see Section 2.1.3), and measure the power draw of the server. This subsection explains the operation logic of the client, starting with the workload definition, followed by the client-side processing of the workload, the power measurement and the workload creation.

### Workload Format

In order to cause different utilization levels of the executor, the client includes some delays between the single jobs of the given workload. However, the concrete specification (i.e., the length and frequency) of the delays could cause variations between sessions, and consequently different measurement outcomes, which is contradictory to a benchmark. In order to avoid this randomness, the client expects the workload to be in a particular format (see Production ⟨workload⟩ of Grammar 3.1), which explicitly specifies the delays to be made.

| | | |
|---|---|---|
| ⟨workload⟩ | ::= | ⟨package⟩ |
| | \| | ⟨package⟩ ⟨workload⟩ |
| ⟨package⟩ | ::= | '**#**' ⟨text⟩ ⟨nl⟩ |
| | \| | ⟨number:delay⟩ '**:**' ⟨number:size⟩ ⟨nl⟩ ⟨bytes(size):content⟩ |
| ⟨number⟩ | ::= | ⟨digit⟩ |
| | \| | ⟨digit⟩ ⟨number⟩ |
| ⟨digit⟩ | ::= | '**0**' \| ... \| '**9**' |
| ⟨nl⟩ | ::= | *the newline character ascii(10)* |
| ⟨text⟩ | ::= | *any text excluding the newline character ⟨nl⟩* |
| ⟨bytes(n)⟩ | ::= | *n consecutive bytes* |

**Grammar 3.1** – Workload file format, starting with ⟨workload⟩

Production ⟨workload⟩ is a sequence of ⟨package⟩s, of which each is either a comment (first rule) or a delay–size pair (second rule) followed by job data. The workload file format is primarily textual, hence each package contains a newline character (⟨nl⟩) as mandatory header separator and the numbers are textually decimal encoded (see ⟨number⟩). The delay of the ⟨package⟩ is specified in microseconds and the size is given in bytes.

The reason for choosing a textual format is that the first job data used were textual, and such a format allows for easy inclusion of debugging comments, hence, makes textual format a convenient choice. However, the job data content is counted in bytes and not determined by any internal syntax, thus allowing non-textual binary-encoded data to be contained.

The generator is application and job agnostic, in spite of the data which is job specific. This is possible because the size of the data is specified and the generator just needs to forward that data to the executor. This keeps the algorithm of the generator compact, as shown in the next paragraph.

### Job Transmission

The processing of the workload file by the generator is shown in Algorithm 3.1. The generator expects the workload to be sent as input stream $W$ in the format specified by Production ⟨workload⟩ of Grammar 3.1, and as output, a stream connection $E$ to the executor. It loops as long as there remains data in the stream $W$. In each iteration the generator reads one line of text into the local variable $L$ (l. 2), which represents the first part of both ⟨package⟩ rules. In line 3, it distinguishes between the two rules—a comment and a job package. Comments are ignored (l. 4), while for all other lines, the header each read package is split into the delay $D$ and the size $S$ (l. 6), both parsed for their textual represented integers defined by ⟨number⟩. Line 7 completes the ⟨package⟩ production by reading the final $S$ bytes of the rule into $C$, which allows the next iteration to continue

```
Require: W ← Input stream from workload file
Require: E ← Output stream to executor
 1: while not end of stream of W do
 2:   Read from W next line L
 3:   if L starts with '#' then
 4:     Discard L
 5:   else
 6:     Split L at ':' into D and S
 7:     Read from W next S bytes C
 8:     Wait D microseconds
 9:     Write C to E
10:   end if
11: end while
```

**Algorithm 3.1** – Generator algorithm

with the next ⟨package⟩. In order to achieve the intended executor utilization level, the generator waits the specified $D$ microseconds (l. 8) before finally sending the data $C$ to the executor $E$ (l. 9).

A further consideration regarding the comparability of the generated data stream is the precise timing of the transmission. Using relative wait calls as the command in line 8 seams to be, has the strong drawback of accumulating the system noise regarding additional delays resulting from the wait invocation itself as well as the read and write operations. Therefore, the actual implementation saves the absolute timestamp at the start of the session (i.e., $t_0$ in Figure 3.6) and adds the given pauses to that timestamp resulting in the absolute point in time to wait for (i.e., $t_i$). This method effectively eliminates accumulative delays, however short-term delays may remain. Also, if the executor can not keep up with the amount of jobs, the generator will be slowed down as well, since the sending of the jobs via TCP will eventually block until the executor reads some of them.

#### Power Measurements

The presented algorithm for the client does not include the power measurement despite being a component of the client. The reason for that is that the presented generator does not carry out the measurement. Instead, the client relies on an external utility, which utilizes an external power-meter device. However, the generator, as presented in Figure 3.6, determines the start timestamp $t_0$ and the end timestamp $t_{end}$, both being important for a precise measurement. Therefore, the generator provides these timestamps to the power-measurement component, which initiates and terminates the measurement accordingly, or extracts the results within this time frame from a possibly long measurement. These two timestampes also mark the two reference points for determining the duration of the processing.

#### Workload Creation

The final yet unaccounted piece on the client side of the prototype is the origin of the workload files. As already mentioned, the generator expects such a file in a specific format and in advance. However, unlike the generator itself, the creation of this file is job specific. Hence the input creator is the only module on the client side which is job specific. Furthermore, it is the only module of the entire prototype, which can be and has to be executed in advance, hence the workload input creators for the jobs are separate executables.

### 3.2.4 Server

The server accepts sessions from clients and their jobs. This functionality is provided by the executor module, however, the executor does not, unlike the the name suggests, execute any job. It rather functions as the link between the client provided jobs and the applications. This way, each application can build on this facility of the executor and does not need to handle the clients itself. This subsection first presents the algorithm of the executor and continues with its API for job processing applications.

#### Executor

The name originates from the concept implemented by the executable containing the executor. As mentioned before, the prototype is intended for rapid interactive workloads. In order to optimize the performance of each job's execution, the server has all the applications build into its binary. This prevents the executor from forking and execute external programs and thus reduces the delay between job processing. Therefore, the server is created as a sort of fat binary, containing various applications, however, unlike the purpose of the previous introduced fat-binary approach, which supplies various equivalent implementations for different platforms, here the fat binary supplies different applications for the same host, the host might still contain heterogeneous components.

**Require:** Server socket $S$
**Require:** Set of applications $Apps$
1: **loop**
2: Accept from $S$ client $C$
3: Read from $C$ the `StreamInit` $I$
4: Select application $A$ out of $Apps$ by ID $I_{app}$
5: Initialize $A$
6: **while** valid connection to $C$ **do**
7: Read from $C$ the `StreamPackage` $P$
8: Forward data $P_{content}$ to $A$
9: **end while**
10: Destroy $A$
11: **end loop**

**Algorithm 3.2** – Executor algorithm

The algorithm of the executor is outlined in Algorithm 3.2. The executor expects a valid server socket $S$ and the set of available applications $Apps$. To serve more than a single client, the executor runs within an endless loop with each iteration handling one session. The executor starts a session by accepting a client (l. 2) and sending the `AWAIT_START` status code. The client answers with a `StreamInit` message (l. 3), which contains the ID of the application to use. If the server knows this application (i.e., it is contained within $Apps$), the executor selects it for the current session (l. 4) and calls the initialization method for that application (l. 5). The executor enters a while loop to process the stream of incoming packages. This loop reflects the loop of the client (see Algorithm 3.1) over its workload, consequently they have the same amount of iterations. In each iteration, the executor reads a package from the client (l. 7) and gives it to the active application (l. 8). Eventually, when the client reaches the end of its workload, it will send the `FINI` status code to the executor, which ends the while loop and finalizes the application (l. 10).

### Application Interface

The executor relies on a given set of applications to successfully process a job. As earlier mentioned executor does not know the specific applications, consequently, the executor has to use a generic interface to communicate with them—an API, which is presented in Listing 3.2.

```
1  struct Task {
2    size_t  size;    // the number of bytes with in the task data
3    char   *data;    // the pointer to the first byte of the task data
4  };
5
6  struct Application {
7    // initializes the environment for one session
8    bool  (*init)    (char* mode);
9    // processes one task
10   void  (*process) (struct Task packet);
11   // cleans up the environment after a session
12   void  (*fini)    (void);
13 };
```

**Listing 3.2** – Application interface

The interface consists of the two structures `Task` and `Application`. `Task` is intended to enclose the job-specific data from the workload, thus it has a `data` field and `size` field to specify the number of bytes in the `data` field. The actual communication is done via the functions provided by the `Application` structure, each represents exactly one application. It consists of three function pointers: 1) `init` prepares the represented application for one session (e.g., initializing OpenCL structures), 2) `process` takes one job and executes the intended algorithm, 3) `fini` cleans up the session (e.g., free structures, allocated in the `init` function).

The decoupling of executor and application is done by:

(1) The executor and each application is placed in its own compilation unit. Thus the not all applications have to be compiled, for example if an application can not be compiled because it requires an API not available for the target.

(2) Each application defines its application structure as a public symbol with a unique name. All other symbols, including the functions referenced in the application structure, should be declared local (i.e., static in C).

(3) The executor declares the total set of the applications as weak symbols (see Section 2.4.3). Declaring the applications as weak symbols, prevents the necessity that they are all defined, allowing to link the server without the requirement to include all of them.

(4) At runtime, the executor tests which of those weak symbols are actually provided (e.g., if the address of such a symbol is non zero).

Listings 3.3 and 3.4 show an example using weak symbols. The first listing shows the implementation of the dummy application, consisting of an init, process, fini function, and the application structure with the symbol `app_dummy`. Listing 3.3 defines an example entry function `main` which uses the weakly declared symbol `app_dummy`. It is important to first test the address of `app_dummy` against zero (l. 4–5) prior to using it, otherwise the program would fail uncontrolled instead of handling the absence of dummy properly. A proper test for the availability of an application is

```
1  static bool init(char* mode) {
2    printf("[Dummy] init\n");
3    return true;
4  }
5
6  static void process(struct Task job) {
7    printf("[Dummy] process job\n");
8  }
9
10 static void fini() {
11   printf("[Dummy] fini\n");
12 }
13
14 struct Application app_dummy = {
15   .init    = &init,
16   .process = &process,
17   .fini    = &fini,
18 };
```

**Listing 3.3** – Example implementation of the application interface

```
1  extern struct Application app_dummy __attribute__((weak));
2
3  int main() {
4    struct Application *app = &app_dummy;
5    if (!app) {
6      printf("No such app!\n");
7    } else {
8      app->fini();
9    }
10   return 0;
11 }
```

**Listing 3.4** – Weak application declaration

relevant, because the executor is part of the server component, which should not crash because of a single client requesting an unavailable application.

This section presented the implementation of a program which is able to evaluates different jobs on given target hardware, focusing on the energy demand caused by varying levels of utilization of the target platform.

## 3.3 Cluster Manager

The previous section introduced a program for generating a profile database, which is used to operate Limo. Consequently, this section presents a cluster manager which makes use of such a profile database—a prototype implementation of Limo. The prototype specializes in managing batch workloads (i.e., set of long-running jobs), suitable for HPC clusters. It implements the power and price awareness introduced by the concept and is later in this thesis compared to the general-purpose cluster manager SLURM, which is actually used in HPC clusters [Pro17b].

First, the following section presents the overall architecture of the prototype, continued by the explanation of the communication protocol used between the cluster-manager nodes in Section 3.3.2. Finally, Section 3.3.3 presents the algorithm that is implemented by this workload manager.

### 3.3.1 Architecture

According to the concept introduced in Section 3.1, the prototype workload manager consists of three programs (see Figure 3.2): `lrun`, `lmaster`, and `lslave`, which are intended to be executed on user nodes, master nodes, and slave nodes, respectively. `lrun` is the program for job submission, `lmaster` is the central manager program, and `lslave` is the executor for the compute nodes.

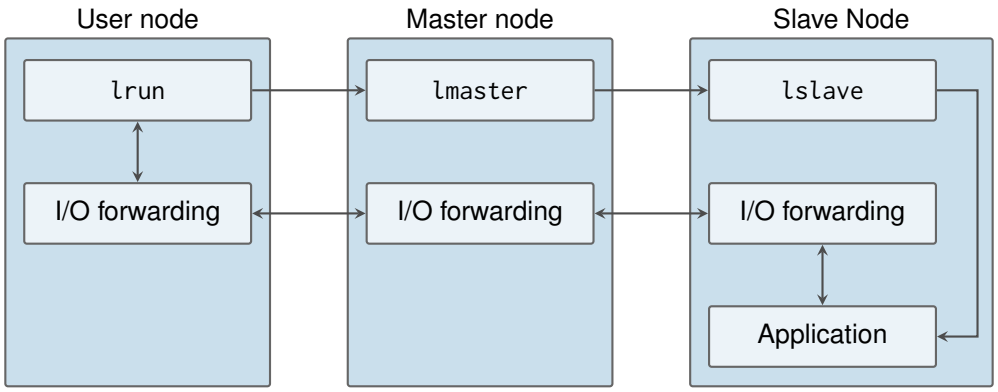



**Figure 3.7** – Workload manager components.

Figure 3.7 shows the three nodes (i.e., machines) of the prototype, which have the following purpose: 1) the user node is controlled by a user who wants to submit jobs via `lrun`, 2) the slave nodes are used to execute those jobs and run an `lslave` instance each, 3) the master node manages the slave nodes and assigns user jobs to an appropriate compute node (i.e., slave node).

Additionally to the three Limo executable, each node consists of an *input/output* (I/O)-forwarding component, which jointly forward the stdin and stdout streams from the user to the slave node and vice versa, respectively. The stderr stream is merged with the stdout stream to avoid the need for a further stream to be forwarded. The slave node contains also the job binaries, which will be called applications, similar to the previous section. However, unlike the applications of the previous section, in this section, applications are standalone executables and not linked into `lslave`.

Having three distinct machines involved within the design, requires communication between them, which is shown in Figure 3.7 by the horizontal arrows. The top line shows the job submit, originating from the user node propagated via the master node to one slave node. The second horizontal lines shows the I/O communication between the user and the running application been relayed over the master node.

### 3.3.2 Inter-Node Communication Protocol

The communication between the three nodes takes place via TCP over IP. The master node opens the TCP server socket to which the others (i.e., `lrun` and `lslave`) connect to. This architecture is the reason that the user and slave have to relay their I/O forwarding over the master—they can not connect directly without establishing further TCP server sockets and connections, which in turn,

would just complicate the design without essential impact on the important aspects of the prototype (e.g., energy-efficient job distribution).

```
1 struct Type {
2   uint32_t type;
3 };
4
5 struct Block {
6   uint32_t size;
7   char data[];
8 };
```

**Listing 3.5** – Workload manager protocol data—`Type` and `Block`

Listing 3.5 defines the basic data structures used in the communication. The communication is message based, each message starting with a `Type` structure that defines the meaning of the message and the data expected to follow. Variably sized data is packaged into a `Block` and a list of strings (e.g., the name of an executable with its command-line arguments) are packed together into one `Block` of null-terminated strings.

The protocol of the Limo prototype consists of exactly six different message types, each represented by one of the following C structures: `NewSlave`, `SendExec`, `DoExec`, `JobAck`, `StreamData`, `TypeConf`. The next paragraphs explain these six message types in detail.

```
1 #define PACKAGE_TYPE_NEW_SLAVE 0x102
2 struct NewSlave {
3   char name[40];
4   uint64_t hardware_bits;
5 };
```

**Listing 3.6** – Workload manager protocol data—`NewSlave`

Whenever a new slave (i.e., `lslave`) wants to join the cluster, it sends the `NewSlave` (see Listing 3.6) message to the `lmaster`. With that message the slave sends a name by which it wants to be referred to, but it also has a debugging purpose in log files since a symbolic name can be better recognized than an IP address or any other numeric ID. Because of this debugging-only purpose, the name is not required to be unique.

Second, the `NewSlave` message contains hardware-specification bits, which indicate what platforms the slave is running (e.g., when component-level heterogeneity is available, for instance a CPU and GPU). These bits are used by the `lmaster` to determine what jobs are appropriate for this slave (e.g., if a job can only be executed on some hardware) and they are used to estimate the power draw and time requirements of this slave for any given job.

The `SendExec` message (see Listing 3.7) is sent by `lrun` to `lmaster` in order to submit a job. The message contains a flag indicating that the job might be delayed or has to be dispatched immediately, which would return an error if not possible. This flag is followed by the packed string of the command-line list (i.e., executable name and argument list) of the job to execute.

The last two fields are optional; that is, they might be an empty list. The member `packed_include_hosts` lists the host names on which the job may be executed. If empty, all hosts are considered for the execution. The member `packed_excluded_hosts` lists the hosts on which the job

```
#define PACKAGE_TYPE_SEND_EXEC 0x101
struct SendExec {
  int8_t is_immediate;
  struct Block packed_commandline;
  struct Block packed_include_hosts;
  struct Block packed_excluded_hosts;
};
```

**Listing 3.7** – Workload manager protocol data—SendExec

may not be executed, and is applied after packed_include_hosts, which means if a host appears in both lists, it is excluded.

```
#define PACKAGE_TYPE_DO_EXEC 0x103
struct DoExec {
  struct Block packed_commandline;
};
```

**Listing 3.8** – Workload manager protocol data—DoExec

DoExec (see Listing 3.8) is similar to the SendExec message and specifies a job to be run, which also contains the packed string of the command-line list. Unlike SendExec, DoExec is send by the master to the selected slave, which just executes it when received.

```
#define PACKAGE_TYPE_JOB_ACK 0x104
struct JobAck {
  // empty
};
```

**Listing 3.9** – Workload manager protocol data—JobAck

However, the master has no information if the slave, which could been idle for a long time, is still active and does really execute the job. Therefore, the lslave is required to confirm the start of execution by acknowledging the receiving of the job with the JobAck message (see Listing 3.9) sent back to the master.

```
#define PACKAGE_TYPE_STREAM_DATA 0x106
struct StreamData {
  struct Block content;
};
```

**Listing 3.10** – Workload manager protocol data—StreamData

In order to avoid an additional connection, the I/O forwarding is tunneled through their normal communication with the StreamData message (see Listing 3.10). That message consists of one Block containing then I/O fowarding data with at least one byte. A StreamData message with a

block size of zero bytes is used to indicated the end of such a tunneled stream, which is send when the application execution terminated.

This stream embedding is not a kind of multiplexing and no normal master–slave communication takes place, which is not required by design. Both the design and the missing multiplexing prevent simultaneous execution of multiple jobs on the same slave, which is reasonable since SLURM by default does not allow this feature (called over-provisioning by SLURM [YJG03]) either, and keeps the comparison fair.

```
1 #define PACKAGE_CHANGE_CONF 0x107
2 struct ChangeConf {
3   uint32_t conf_type;
4   struct Block content;
5 };
```

**Listing 3.11** – Workload manager protocol data—TypeConf

All messages required for the essential operation of Limo have been presented, however, since this workload manager considers especially power limits, which are subject to change, as earlier mentioned, one last message type is introduced to reconfigure the workload manager during execution—the ChangeConf message (see Listing 3.11).

The messaging pattern has the advantage of the possibility to introduce configuration changes from a remote machine. However, a special executable has been introduced for transmitting those messages—lconf. Each ChangeConf message consists of a configuration type number (e.g., set lower power limit, set higher power limit) and the value for this configuration (e.g., the watt value as 32-bit float).

Figure 3.8 shows the message sequence of the communication between the three nodes. The diagram starts with the transmission of the NewSlave message from a slave to the master. Since first message is only sent once and can be done long before any user gets involved, the figure has intermission line drawn there. However, with this first message the slave is registered at the master and available to process jobs.

When users decides to submit a job, they starts lrun giving it the name of the executable and command-line arguments of the job to submit. lrun sends this job specification with SendExec to the master. The master then starts its dispatch routine, which searches an appropriate slave node for the given job. When the dispatch routine found one such node, the master sends the DoExec message to it and waits for the JobAck message. If the connection broke down (e.g., if the slave went offline) or another protocol error occurred (e.g., another unexpected message was received), the master detects this and lets the dispatch routine choose another slave without loosing the job.

When the slave receives a DoExec message it first replies with JobAck and executes the job. During the execution of the job, the slave transmits and receives StreamData messages for the I/O forwarding. Those StreamData messages are extracted at the master and sent as raw data over the connection to the client, analogous, raw data from the client is packaged at the master into StreamData messages for the slave. This extraction and packaging of the I/O forwarding encapsulate it within the master–slave connection, which in turn allows to continue with the protocol between the master and the slave independent of a correct and successful job execution.

When the execution of a job ends the I/O streams cease too, and the slave transmits an *end of file* (EOF) StreamData message (i.e., without any content) to the master, which then marks the slave as available again and the master closes the connection to the user.

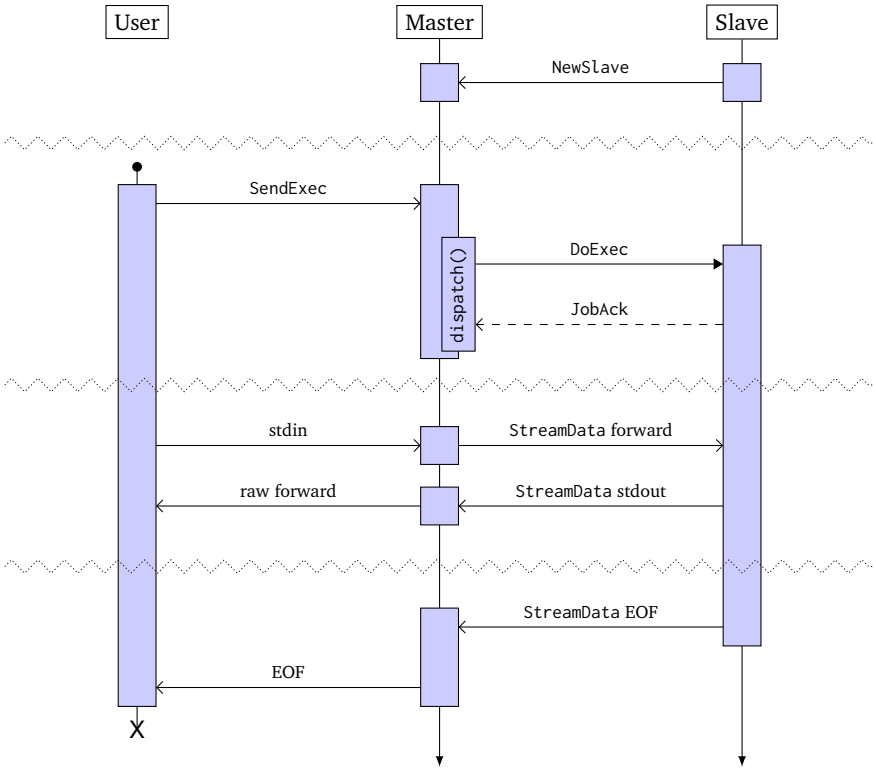



**Figure 3.8** – Workload-manager communication protocol.

### 3.3.3 Workload Manager Algorithm

The job dispatch algorithm of Limo bases on three parameters: a low power limit, a high power limit and the current operation mode. The operation mode defines if Limo should try to reach the upper high power limit or to stay at the low power limit (i.e., utilize more power if electricity is cheap and less power if electricity is expensive). Section 3.1 introduced the grid and the spot-market components, consequently grid component provides the power limits and the spot-market component (see Section 3.1.2), which monitors the electricity price, defines the current operation mode.

The profile database consists of a collection of files within the file system. For each application there is a separate file listing the power and time requirements for the job for each hardware type. The files are named by the first string behind the `packed_commandline` field of the `SendExec` message in Listing 3.7, assuming that the command is a simple name without a path (if there is a path given, it is ignored when searching for a profile file). The hardware type of a slave is specified by the `hardware_bits` in the `NewSlave` message of Listing 3.6.

Placing the profiles of the different applications in different files allows to organize them easier (e.g., creating and removing files). The fully qualified path of an application is ignored in order to increase the stability of the system, also slaves ignore the fully qualified paths and execute applications only in there local working directory, for the same reason.

The hardware bit abstraction simplifies the incorporation of multiple slaves consisting of the same hardware specification. This simplification is appropriated, because the contained values

are averaged values to be used to make an estimate. Further the contained power values state the additional power draw, which depends on the actual executing hardware and less on other components in the system (e.g., an additional graphics card), since there power draw would remain constant when not used.

**Require:** List of jobs to be dispatched $J$
**Require:** Set of available slaves $S$
**Require:** Power limit $(L_l, L_h)$
**Require:** Operation mode $M \in \{high, low\}$
**Require:** Current power draw $C$
**Require:** Profile database $D : (j, s) \rightarrow p$, where $j$ is a job, $s$ is a slave, $p$ is an average power draw

```
1:  while J, S not empty and (M = high or C < L_l) do
2:    Get from J first element j
3:    for s ∈ S do
4:      Get from D(j,s) as d
5:      if M = high and (C + d < L_h or C < L_l) or
        M = low and C + d < L_h then
6:        Dispatch j on s
7:        Remove s form S
8:        if dispatch successful then
9:          Remove j form J
10:         Break for loop
11:       end if
12:     end if
13:   end for
14: end while
```

**Algorithm 3.3** – Job dispatch algorithm of the Limo prototype

Algorithm 3.3 shows the method Limo selects a proper slave for a given job in accordance with circumstances. The algorithm is dispatches as many jobs as possible in one invocation. It is invoked whenever there is a chance of dispatching a new job (e.g., when a new slave or job was received, the limits change, or the execution of a running job has finished). It essentially implements the introduced concept from the beginning of this chapter (see Figure 3.4).

The requirements of the algorithm are the list of jobs, which are submitted but not dispatched $J$. This list is simply maintained, by appending any job received via `SendExec`, and removing any successfully dispatched job.

The second requirement $S$, the set of available slaves, is maintained by adding new slaves (`NewSlave`), removing busy slaves when a job is dispatched to the slave, and adding again slaves, which finished executing a job. If a slave is broken down (e.g., went offline), is detected when the next job is tried to been executed on that slave, and the slave failed to send the `JobAck` message.

As already mentioned, the power limit and operation mode are supposed to be provided by the grid and the spot-market components, respectively, and the current power draw is supposed to be provided by the power-meter component (see Section 3.1.2). However since this a prototype, the current power draw is estimated by the sum of estimated power draws originating form the profile database, and the power limit and operation mode are specified via the `ChangeConf` message.

The final requirement of the algorithm—the profile database—is provided in advance. For the workload-manager prototype the presented profile generator prototype of Section 3.2 can not be used, because this one is specialized in batch workloads and the other is specialized interactive workloads.

Therefore, an less sophisticated but practical approach has been used to obtain a profile database: The whole cluster is measured via an external power meter and each job is executed on each slave one after another, utilizing the blocking nature of the `lrun` command and the `packed_include_hosts` and `is_immediate` fields of the `SendExec` message.

As a result of dispatching as many jobs as possible, Algorithm 3.3 start with a while loop, which continues as long as there are at least one job and one slave in the sets $J$ and $S$, respectively, and Limo is running in high-power operation mode ($M = high$) or the current power draw ($C$) is lower than the low power limit ($L_l$). This condition terminates the algorithm if either no job or no slave is currently available, and it terminates if the low power limit has been passed while in low-power operation mode.

The following line (l. 2) fetches the next job $j$ from the queue. Limo executes jobs (at least the start of execution) in the order of submit. For the current job all available slaves are traversed, and the estimated power draw for the combination of job and slave are retrieved from the profile database (l. 4). The condition in line 5 tests if the current combination obeys the power limits, one may notice that $C$ is the old power draw and $C + d$ is the estimated new power draw. In the following, the current power draw will be called $P_t := C$ and the new power draw will be called $P_{t+1} := P_t + d$. Additionally the while loop gives $P_t < L_l$ if $M = low$.

The if clause maintains

$$L_l \leq P_{t+1} \leq L_h \tag{3.1}$$

in order to obey the power limits using a best-effort approach. The best-effort approach is needed because this equation can not be satisfied in all cases, assume for example

$$P_t + a < L_l, \forall a \in D \tag{3.2}$$

that means the current power draw $P_t$ is so low that there is no combination of job and slave in the profile database $D$ to yield a $P_{t+1} := P_t + d$, which maintains the power limits described by Equation (3.1). Consequently, dispatching any job violates Equation (3.1) and therefore the power limits.

This problem has been discussed in Section 3.1.3, and the proposed solution is the weaken the constraint. One way would be add as precondition that the current power draw already satisfies the power limit, which has the draw back that if the power limit is not satisfied, the power limits would be entirely ignored, possibly changing from violating the lower limit to violating the upper limit.

In order to solve the power limit violation problem by best-effort approach, the following assumptions are made:

(1) In low power operation mode, violating the low power limit is less serve than violating the high power limit, because the power draw should be as low as possible.

(2) In high power operation mode, violating the high power limit is less severe than violating the low power limit, because the power draw should be as high as possible.

Therefore, Equation (3.1) is simplified to Equation (3.3) for the low power operation mode. Consequently, dispatch until the high power limit is exceeded is valid. However, this does not imply that the low power operation mode would usually reach the upper limit, since the algorithm given by the while condition stops in the low power operation mode as soon as the low power limit has been reached.

$$P_{t+1} \leq L_h \tag{3.3}$$

The high power operation mode requires an additional term shown in Equation (3.4). The reason for the implication put in front of the equation is, that the high power operation mode might

violate the low power limit such as in the case of the above counterexample (see Equation (3.2)), where the system draw far too less power. Consequently, the algorithm obeys the high power limit when the limits can be satisfied, which is indicated by the current power draw being above the low power limit $L_l \leq P_t$. Otherwise, for example if the target zone is too narrow, the high power limit may be violated in order to rise above the low power limit.

$$L_l \leq P_t \;\rightarrow\; P_{t+1} \leq L_h \tag{3.4}$$

Continuing with Algorithm 3.3, the dispatch only happens (l. 6) if the estimated new power draw holds Equation (3.3) or Equation (3.4) depending on the operation mode. The dispatch consists essentially of sending the `DoExec` message, after which the slave is marked as busy and removed from the available slave list $S$ (l. 7).

The protocol requires the slave to answer the `DoExec` with `JobAck`, if the slave fails the send the message, the slave is assumed broken and the connection between the master and that slave is terminated. One may notice that the job itself has not been removed for the job list $J$ and the algorithm—in this case the for loop continues with the next available slave. Since the slave did not manage to even send a simple `JobAck` message, it is reasonably safe to give the job another slave for execution. If t the slave successfully send the `JobAck` message, the master assumes the execution of the job and removes it for the job list (l. 9) and continues with the next job by breaking the current for loop (l. 10).

This section presented the workload manager which considers heterogeneous compute nodes and exploiting there different power draws it obeys given power limits on a best-effort base. Additionally, utilizing the two operation modes of this prototype, it is capable of exploiting varying electricity prices. The effectiveness of the prototype will be demonstrated in the next chapter.

## 3.4 Summary

This chapter presented the concept of Limo a heterogeneous-aware cluster manager which exploits this heterogeneous compute nodes and electrical-energy–price awareness in order to enable cost efficient operation, while it obeys given power limits on best-effort basis.

Following this concept, a prototype for evaluating the potential of heterogeneity on system with an interactive workload, which can be used to generate profile database for the Limo concept.

Finally, a prototype implementation of Limo was presented which implements heterogeneity awareness and power awareness, which allows it to stay within given power limits on best-effort base. The remaining degree of freedom is restrict due the introduction of two dynamic operation modes, which can be used to exploit varying electricity prices.

In the following chapter, the two prototypes are evaluated and the workload manager is compared to the SLURM workload manager.

# 4 EVALUATION

This chapter presents the evaluation of the previously discussed prototypes. The first section introduces the evaluation setup (i.e., the hardware and the software used). In the following, three different evaluation experiments are presented: 1) the results of running a benchmark on a single graphics board at various frequencies is presented. 2) the results of running the profile generation prototype (see Section 3.2) on different hardware and at different power management settings (i.e., frequencies and power caps) is shown. 3) the results of comparing the Limo prototype (see Section 3.3) with SLURM is presented.

## 4.1 Hardware and Sofware Setup

This section presents the hardware and software (i.e., the benchmark which have been evaluated) used in this evaluation. Starting with the hardware which has been used. Followed by the presentation of the used software and benchmarks.

### 4.1.1 Hardware

Figure 4.1 shows the overall assembly of hardware, which has been used in the following tests. Only parts of the shown components were used for the first and second part in this evaluation. The last evaluation part utilizes the entire shown setup.

| Device | Processor details | Memory |
|---|---|---|
| Ⓐ Hardkernel ODROID-C1+ | Amlogic Cortex-A5 (4x 1.5 GHz) | 1 GiB |
| Ⓑ Hardkernel ODROID-C2 | Amlogic Cortex-A53 (4x 2.0 GHz) | 2 GiB |
| Ⓒ Fujitsu CELSIUS W550 | Intel Xeon E3-1275 v5 (8x 3.6 GHz) | 16 GiB |
| Ⓓ Intel HD Graphics P530 | 24 pipelines (1.15 GHz) | shared |
| Ⓔ Nvidia Quadro P2000 | 1,024 CUDA cores (1.375 GHz) | 5 GiB |

**Table 4.1** – Overview of the evaluated devices with system-scope (A–C) and component-scope (D–E) heterogeneity [Hön+18].

Table 4.1 lists the different computation hardware, which are shown and marked in the lower picture of Figure 4.1:

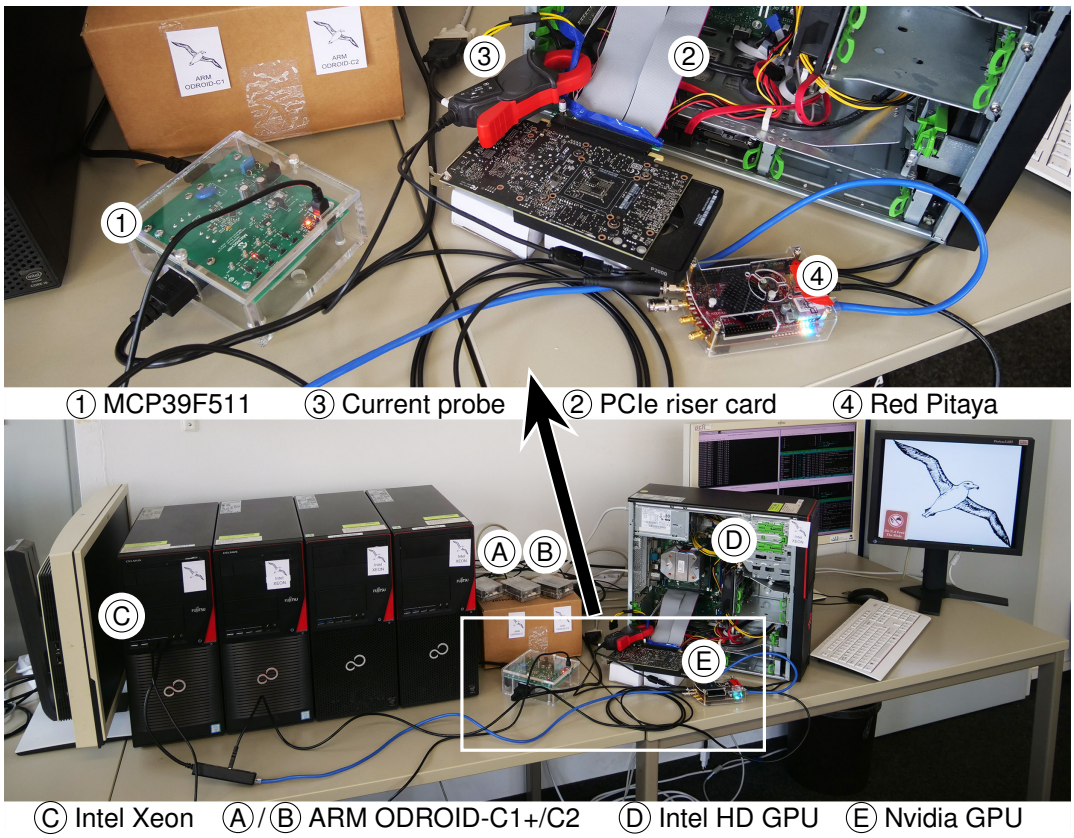



**Figure 4.1** – The cluster setup of the evaluation [Hön+18]. The upper picture shows the graphic board surrounded by the measurement instruments. The lower picture shows an overview of the compute nodes of the cluster.

Ⓐ **ODROID-C1+** The first of the two ODROID *single-board computer* (SBC) used. It is the slowest hardware type in the evaluation but it has also the lowest power draw, which makes it an interesting platform for a power-aware cluster manager.

Ⓑ **ODROID-C2** The second ODROID board which is an successor of the ODROID-C1+ and supersedes it by its capability. However, the increased capability of the ODROID-C2 comes with a higher power draw.

Ⓒ **Xeon** A non-consumer high-end CPU from Intel. It is used to compare the other system against CPU processing. It worth noticing that the Xeon CPU is significantly more powerful (regarding compute capability) and has and significant higher power draw than the ODROID systems.

Ⓓ **Intel HD** The first GPU of the evaluation. It is integrated into the Xeon CPU. Such integrated GPUs are less powerful than graphic boards, but are shipped together with most CPU and, therefore, available on a wide range of systems (e.g., the ODROID systems have also an integrated GPU).

Ⓔ **Quadro** The second GPU is the dedicated Nvidia Quadro P2000 graphics board, which has a significant amount of parallel processing power compared to the Intel HD and has a much higher power draw.

Beside the computation hardware a set of measurement devices were also used. First of all the MCP power meter ① which was used in all evaluation parts. The other three devices ② – ④ were used to examine specifically the power draw of the Quadro Ⓔ. The detail of these four devices:

① **MCP power meter** The MCP39F511 is a power meter from Microchip, which allows to read out the current power draw via *universal serial bus* (USB) at sampling rate of 50 Hz and a measuring error of $\leq$ 0.1 % [Mic15]. It is capable of monitor the power supply, which makes it the prefect tool to read the total system power draw.

② **PCIe riser card** This connector is essentially an extender cable for the PCIe bus. It was used to make the power supply of the Quadro graphic board, which is included in the PCIe lane accessible for measurement.

③ **Current clamp** The MH60 current clamp from Chauvin Arnoux. It measures the current flow through the wires enclosed by its clamp exploiting the Hall effect. It is has signal bandwidth of 1 MHz [Cha17].

④ **Red Pitaya** The STEMLab Red Pitaya board in the 1.1 version [Ste18]. It is used to receive the high frequency signal from the current clamp and calculate the power draw of the graphics board.

The results of the power measurement of the Nvidia Quadro showed, that the internal measurment of the GPU, which is accessible via the `nvidia-smi` tool are reasonably accurate under all measured levels of utilization.

### 4.1.2 Software

The most important software used are the two prototypes introduced Sections 3.2 and 3.3. Beside those prototypes several benchmarks have been evaluated, which were taken from two independent benchmark collections: the Rodinia suit [Che+09] and the *National Aeronautics and Space Administration (*NASA*) advanced supercomputing division (*NAS*) parallel benchmarks* (NPB) [Bai+91].

#### LUD

The first two experiment of the evaluation examine benchmark of the Rodinia suit—namely the kmeans and *lower upper decomposition* (LUD). However, the presented results are obtained only using LUD, since the kmeans result were partially incomplete and did not show significant deviations worth further investigation.

LUD takes an arbitrary sized quadratic matrix and calculates the lower-upper decomposition (also called factorization) of the matrix, which is used in order to calculate the inverse matrix. Formally, LUD gets a matrix $A$ and calculates the triangular matrices $L$ and $U$ in order that $A = L \cdot U$ is true. However, the algorithm is implemented using a $16 \times 16$ sub matrix subdivision and hence the input matrix dimension must be a multiple of 16.

The Rodinia suit provides the LUD benchmark in a variety of implementations. The three implementations using OpenMP, OpenCL, and CUDA, respectively, where chosen to be included in this evaluation. However, the CUDA version was taken form a third party implementation from Hangchen Yu [Yu+17] while the other version originate form the official implementation [WCS15].

All three implementation require an invertible matrix (i.e., a matrix which has an inverse) as input. Unfortunately, a invertible matrix can not be generated simply by random numbers, instead it the implementations construct a invertible matrix by generation two triangular matrices like the $L$ and $U$ matrices by random numbers and multiply them.

However, this matrix generation induces unnecessary overhead into the benchmark and the resulting matrix differs each run, therefore the implementation can alternatively take a textual encoded matrix generated in advance. Nevertheless, the decoding of the matrix causes still a significant overhead. This overhead is specifically interfering with the evaluation, because the decoding is carried out by the CPU, which biases the results when comparing the CPU with an GPU. This effect is shown as a by-product in Section 4.2.

In order to reduce the overhead of the LUD benchmark, the prototype of Section 3.2 stores its LUD input in a binary format.

#### NPB

The workload manager from Section 3.3 targets an HPC environment and is following compared with a general-purpose HPC workload manager. In order to set the evaluation also into an HPC environment a different benchmark set, which is widely adopted in this domain, the NPB are used for that evaluation.

| Benchmark | Long name | Description |
|---|---|---|
| CG | Conjugate Gradiant | Estimate the smallest eigenvalue of a large sparse symmetric positive-definite matrix using the inverse iteration with the conjugate gradient method as a subroutine for solving systems of linear equations. |
| EP | Embarrassingly Parallel | Generate independent Gaussian random variates using the Marsaglia polar method. |
| FT | Fast Fourier Transform | Solve a three-dimensional partial differential equation using the fast Fourier transform. |
| IS | Integer Sort | Sort small integers using the bucket sort. |
| MG | MultiGrid | Approximate the solution to a three-dimensional discrete Poisson equation using the V-cycle multigrid method. |

**Table 4.2** – Description of the benchmarks used in the workload manager evaluation [Bai+91].

Table 4.2 lists the five *kernel applications* of the benchmark set, which are the five applications used in this evaluation. In the context of the NPB are three further so called *pseudo applications*, but those have not been considered in this evaluation.

The five kernel benchmarks are supposed to reflect the different computation tasks required in a wide domain of HPC applications. For the evaluation all five programs are used together and each with the same frequency.

Opposing to the Rodinia suit, the NPB have unified problem size hard coded into the programs, which are called *benchmark classes*. Each class is identified by a single character and order in increasing computation expense [Dun16]:

**S** smallest size, intended for test purpose.

**W** workstation size (assuming a 90's workstation)

**A, B, C** standard test problem sizes

**D, E, F** large test problems

The problem size A showed an adequate execution time for the purpose of this thesis on the given cluster, hence only size A of the five benchmarks is used in this evaluation.

In order to execute the benchmarks on the evaluated GPUs, an implementation in OpenCL is required, however the official NPB web page lists no such implementation [Dun16]. Therefore, the unofficial OpenCL implementation of the NPB from Sangmin Seo, Jungwon Kim et al. from the Seoul National University have been chosen [Kim+12; Seo+13].

## 4.2 GPU Frequencies

In the first experiment of this thesis the Nvidia Quadro P2000 Ⓔ executes the LUD and kmeans benchmark from the Rodinia suit in the CUDA implementation at varying core frequencies of the GPU, and different job sizes (e.g., matrix dimensions). The power draw of the system and duration the execution is measured and compared between the different core frequencies.

The goal of this experiment was to analyse the energy efficiency of the different GPU frequencies. However, the benchmark is not suited to examine the energy efficiency of its execution. And only the results of LUD benchmark and huge matrix size are shown to visualize the occurred issue.

The given Nvidia GPU supports configuration via the `nvidia-smi` tool, which is a Linux program that communicates with the GPU driver and allows to readout internal values (e.g., temperature, utilization, power draw) and to reconfigure setting of the GPU such as the clock frequency (called *clocks.applications* by `nvidia-smi`). This clock frequency is two fold, consisting of the memory clock frequency and the graphics clock frequency.

Both frequencies have to be set together, however the configuration space of 290 valid settings of the Quadro P2000 allows to nearly independently choose the graphics clock and memory clock. The memory allows only three different values: 405 MHz, 810 MHz and 3504 MHz, whereas the graphics clock allows a range of 139 MHz–1721 MHz in 12.66 MHz steps (as integers of MHz). However, if the memory is set to 405 MHz the maximum frequency for the grapics clock is 607 MHz.

Figure 4.2 shows the impact for the 290 frequency configurations of the Quadro on the execution duration of the LUD benchmark with a matrix size of $16384 \times 16384$. The x-axis shows the frequency of the graphics clock, the y-axis shows the total duration of the execution. There are four graphs plotted into the figure.

The first three graphs (solid lines) show the impact of the graphics clock frequency on the execution duration, each for a different the memory frequency. Since the three lines have only a mean deviation of 0.38 s from each other, it can be assumed that there is no impact of memory frequency on the execution duration.

The observation of the missing memory frequency impact is no measuring error, because further investigation of the matter using `nvidia-smi` revealed that the actual GPU memory clock (called clocks.current.memory) is always on 3504 MHz when the GPU is utilized despite any specified memory clock (called clocks.applications.memory) which is displayed to be set to the requested value. Therefore, the missing effect of the memory clock frequency is either a bug in `nvidia-smi`, the Nvidia GPU driver or the GPU itself.

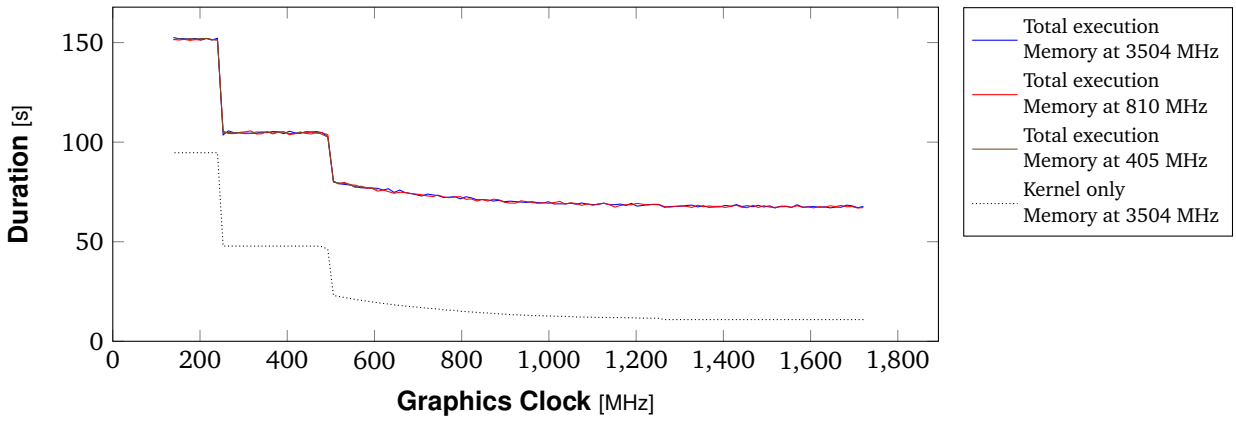



**Figure 4.2** – Impact of all configurable GPU frequencies on a Nvidia Quadro P2000 execution LUD of a 16384 × 16384 Matrix—Execution Duration

Another observation of the three graphs is that the graphics clock frequency has an overall expected impact on the execution duration; that is, the lower the frequency is the longer takes the execution. However, variation of the frequency does not have a continuous impact on the execution duration. The are discontinuities in the execution duration above 240 MHz and 493 MHz, and constant duration areas between 139 MHz and 240 MHz, 253 MHz and 493 MHz, and 1265 MHz and 1721 MHz. Only in the remaining space from 506 MHz to 1265 MHz, an continuous effect of the changing graphics clock can be observed.

Finally, the last graph (dotted line) in Figure 4.2 shows the actual execution duration of benchmark, the so called kernel, which is executed on the GPU. Though the kernel takes on average 56.90 s less than the total execution time, which is a significant offset, and indicates a massive overhead. Furthermore, the overhead is not impacted by the change of the GPU frequencies.

Investigation of this anomaly revealed that the additional time is spend by the CPU instead, and before the GPU executes the actual benchmark routine. The source code revealed that the extra time was caused by reading the input matrix from a textual encoded file. The utility function `create_matrix_from_file` of the benchmark, which read a matrix (i.e., array of floats) form a file, contains the following lines; `size` denotes the dimension of the matrix, `fp` the input file pointer, and `m` the matrix array:

```
1 for (int i=0; i < size; i++) {
2   for (int j=0; j < size; j++) {
3     fscanf(fp, "%f ", m + i*size + j);
4   }
5 }
```

In the present evaluation, the 16384 × 16384 matrix causes the above `fscanf` function to be invoked over 268 million times and each decodes one float value from text, which in this evaluation, took roughly 57 seconds. However, this was not intended to be evaluated, and the reason that the prototype evaluated in the next section does not decode its input from text but instead reads the matrix as one *binary large object* (BLOB) and casts it into a float array.

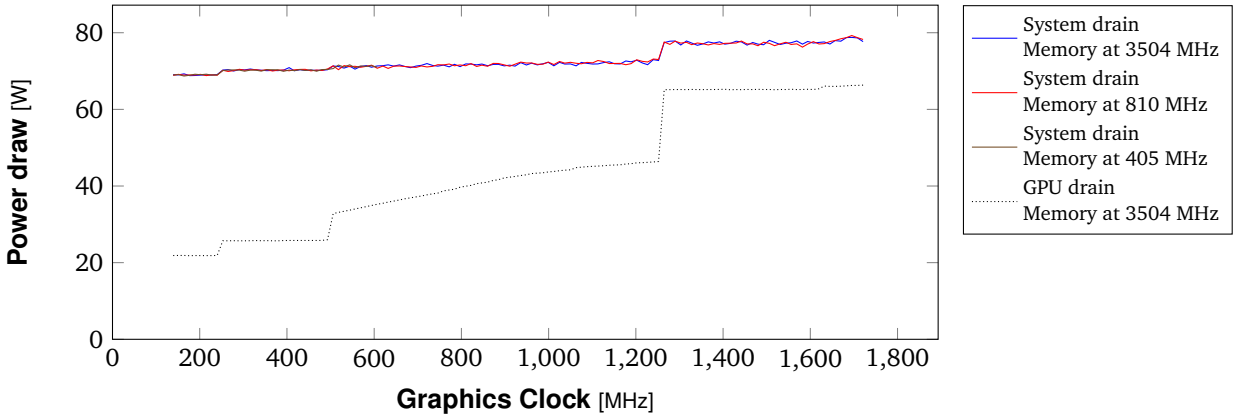



**Figure 4.3** – Impact of all configurable GPU frequencies on a Nvidia Quadro P2000 execution LUD of a 16384 × 16384 Matrix—Average Power Draw

Figure 4.3 shows the power draw during the same benchmark execution. The x-axis shows again the frequency of the graphics clock, and the y-axis shows power draw. There are also four graphs plotted into the figure.

The first three graphs (solid lines) show this time the impact on the average power draw of the system. The memory frequency has no impact on the power draw either, and the graphics clock frequency has only a slight impact on average power draw.

Nevertheless, three discontinuities in the power draw curve are visible, the first two discontinuities just above 240 MHz and 493 MHz are at exactly the same frequencies as the duration discontinuities. The third and most significant discontinuity is just above 1265 MHz, which is the still a noticeable frequency in Figure 4.2. One may assume that the duration discontinuities at this frequency too, but too less to be noticeable.

The last graph in the figure shows the maximum power draw reached by the GPU. Interestingly, this graph shows that the graphics clock frequency acts indeed as a power cap on the GPU, because at lower frequencies the power draw of the Quadro peaks at a significantly lower level. This graph shows surprisingly even a forth power draw discontinuity just about 1620 MHz, also this graph shows the continuous impact between the same frequencies as in Figure 4.2 form 506 MHz to 1265 MHz.

However, it might be assumed that this peak power draw shown in the last graph is the proximate power draw of the GPU during the execution. The reason that its significant increase in power draw does only slightly affect the average system power draw is two fold: 1) the already discussed overhead applies here too, causing a strong background power draw, which reduces the effect of the power draw during the kernel execution on the system power draw, 2) the increased power draw at higher frequencies correlates with shorter execution duration, which further reduces the effect on the system power draw.

The final figure of this experiment (Figure 4.4) shows the efficiency of the evaluated benchmark, regarding the benchmark goal. As mentioned, the evaluated LUD benchmark is not suited to examine energy demand, because it has an significant input overhead (i.e., reading the matrix), compared to the actual benchmark routine (i.e., decomposing the matrix).

The x-axis of the figure shows still the graphics clock frequency, and the y-axis shows a proportion. In this case the only graph in the figure shows the share of the kernel execution on the entire benchmark duration, which is quite low. It peaks at the lower frequencies at 63 % and declines down to 16 % at the higher frequencies. The average is 26 %, which is too low to gather reliable energy

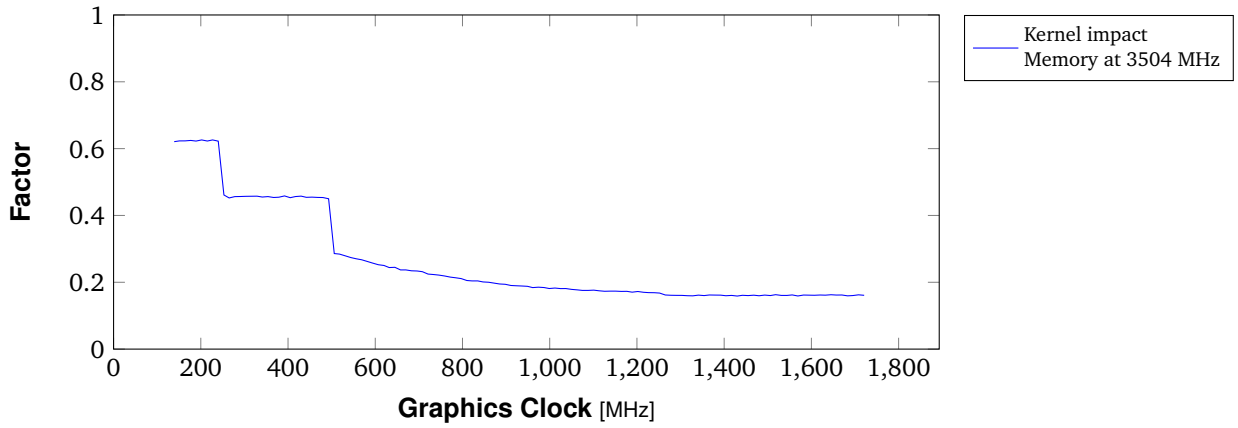



**Figure 4.4** – Impact of all configurable GPU frequencies on a Nvidia Quadro P2000 execution LUD of a 16384 × 16384 Matrix—Benchmark efficiency

demand data. The value is impact by the graphics clock, because it increases the kernel execution time while the overhead stays constant, hence it the efficiency higher at the lower frequencies where the kernel takes the most time.

To summaries this section the change of the graphics clock frequencies has an impact on the execution duration as well as power draw of the Quadro, but the high overhead of the evaluated benchmark makes the comparison of the energy efficiency among the different frequencies unreliable.

## 4.3 Interactive Workload Characteristics

The workload generator presented in Section 3.2 has also the ability to show the potential of the Limo concept by evaluating the differences in energy demand among the different compute hardware components, which is the goal of the second experiment of this thesis.

This section presents the result of running 20 different workloads on three different compute nodes (Xeon Ⓒ, Intel HD Ⓓ, Quadro Ⓔ) in three different software implementations (OpenMP, OpenCL, CUDA) of the LUD benchmark from the Rodinia suit. However not all implementation could be run on every hardware, CUDA for instance runs only on Nvidia graphic boards such as the Quadro, resulting in 5 different combinations of hardware and benchmark implementations—the *targets*: OpenMP on the Xeon, CUDA on the Quadro, and OpenCL on all three.

Additional, the effect of RAPL power caps and different GPU frequencies were evaluated by running the benchmarks with different power management configurations (i.e., power caps and GPU frequencies). Seven different power caps (8 W, 12 W, 16 W, 22 W, 30 W, 40 W and 60 W) plus no power cap, and three different GPU graphics clock frequencies (810 MHz, 1075 MHz and 1240 MHz) were evaluated.

Table 4.3 lists the 20 LUD workloads evaluated in this experiment, all have an goal duration of roughly 60 s. They vary in job size (i.e., the dimension of the matrices) and computational throughput, and are grouped by there job size (second column).

The workloads are randomized and consist of matrices with varying dimension. The name of each workload states the goal dimension and the second column of the table the actual average matrix dimension.

| Name | Matrix size | Matrices | Job size[a] | Throughput[a] | Duration |
|---|---|---|---|---|---|
| m16_i0 | 23 × 23 | $2.22 \times 10^6$ | 33.5 kOp | 1.21 GOp/s | 61.3 s |
| m16_i10 | 24 × 24 | $2.03 \times 10^6$ | 33.5 kOp | 1.12 GOp/s | 60.4 s |
| m16_i50 | 24 × 24 | $1.49 \times 10^6$ | 33.5 kOp | 819 MOp/s | 60.7 s |
| m16_i200 | 24 × 24 | $743 \times 10^3$ | 33.5 kOp | 409 MOp/s | 61.0 s |
| m64_i0 | 79 × 79 | $221 \times 10^3$ | 2.54 MOp | 9.16 GOp/s | 61.3 s |
| m64_i10 | 79 × 79 | $202 \times 10^3$ | 2.54 MOp | 8.38 GOp/s | 61.3 s |
| m64_i50 | 80 × 80 | $148 \times 10^3$ | 2.55 MOp | 6.13 GOp/s | 61.3 s |
| m64_i200 | 80 × 80 | $73.6 \times 10^3$ | 2.56 MOp | 3.07 GOp/s | 61.3 s |
| m256_i0 | 319 × 319 | $15.1 \times 10^3$ | 178 MOp | 43.7 GOp/s | 61.4 s |
| m256_i10 | 320 × 320 | $13.7 \times 10^3$ | 178 MOp | 39.9 GOp/s | 61.4 s |
| m256_i50 | 320 × 320 | $10.1 \times 10^3$ | 179 MOp | 29.3 GOp/s | 61.3 s |
| m256_i200 | 318 × 318 | $4.99 \times 10^3$ | 177 MOp | 14.4 GOp/s | 61.4 s |
| m1024_i0 | 1285 × 1285 | 943 | 11.9 GOp | 182 GOp/s | 61.6 s |
| m1024_i10 | 1297 × 1297 | 846 | 12.1 GOp | 165 GOp/s | 61.7 s |
| m1024_i50 | 1303 × 1303 | 627 | 12.0 GOp | 123 GOp/s | 61.5 s |
| m1024_i200 | 1268 × 1268 | 336 | 11.0 GOp | 60.1 GOp/s | 61.7 s |
| m4096_i0 | 5323 × 5323 | 21 | 914 GOp | 284 GOp/s | 67.7 s |
| m4096_i10 | 4848 × 4848 | 24 | 729 GOp | 254 GOp/s | 68.9 s |
| m4096_i50 | 5426 × 5426 | 16 | 813 GOp | 190 GOp/s | 68.3 s |
| m4096_i200 | 4546 × 4546 | 11 | 520 GOp | 85.1 GOp/s | 67.2 s |

[a] The number of operations are calculated using Equation (4.1). *Op* is used as unit for operations.

**Table 4.3** – Specification of the workloads used in the evaluation Section 4.3.

The first workload of each group (i.e., those with name suffix i0) contain that many matrices that at least one of the targets could execute it in just the specified duration. However, some of the i0 workloads do not utilize all targets at full capacity, because the used prototype has to transfers the workload over the network to the target and some workloads are limited by the network throughput of the installed Gigabit Ethernet (1000BASE-T) connection with 1 Gbit/s. The i0 workloads have a size of 5.6 GiB–6.5 GiB, causing network traffic of 91 MiB/s–111 MiB/s.

The other workloads in each group have a reduced amount of matrices compared with the respective i0 workload. The workloads i10 have 91 %, i50 have 67 %, and i200 have 33 % the matrices of the respective i0 workload.

Finally, for better comparison of the workloads an empirical metric has been defined, which estimates the computational expense of an LUD job in operations. It was derived from the analysis of the source code of the OpenMP implementation. The metric is defined as following, $N_d$ is the estimated number of operations to decompose a matrix with the dimension $d$.

$$N_d = 16896 \cdot d^3 - 21120 \cdot d^2 + 6824 \cdot d \tag{4.1}$$

Using this metric, the Table 4.3 shows the estimated average operation count per job (i.e., a matrix) of the workload and the average computational throughput caused by the processing of the workload in operations per time.

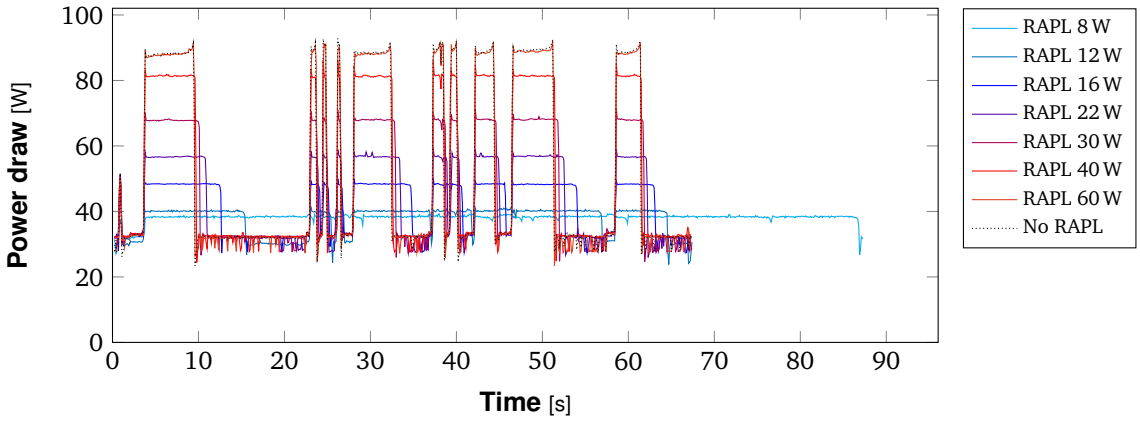



**Figure 4.5** – The effect of DVFS on CPU via RAPL. Shows OpenCL on Xeon executing m4096_-i200.

Figure 4.5 shows the power draw variations of the Xeon total system power draw during the execution of the m4096_i200 workload using the OpenCL implementation at different CPU power caps. The x-axis shows the time-line of the execution and the y-axis shows the power draw of the entire system.

The first observation is that the power draw is not constant, is this case 11 power peaks and one small peak at the beginning of the graph are visible as almost all graphs, notice the notch at 38.2 s splits the apparent peak 37.2 s–38.7 s into two. These 11 peaks correspond to the 11 matrices in this workload, each one represents the processing of one matrix by the Xeon. In the mean time between two such matrices, the Xeon is just idle, which is not uncommon for systems with interactive workloads, and hence draws less power.

The small peak in the beginning at 0.9 s is the typical overhead of OpenCL, compiling the textually given kernel code, which is the actual processing routine. This is outstanding among the evaluated implementations, since OpenMP is directly compiled into the binary, without the need of an run time compilation step, and CUDA which has an similar concept of an separate kernel code, uses so call PTX code—an assembly code for faster compilation and so called cubin object code in order to avoid the run time compilation entirely [Nvi18].

The graphs show further that all matrix processing peaks (if the preceding already ended) start at exact the same point in the respective time-line, which shows that the delay timing of the workload is implemented as intended. Also the processing duration of the different peaks in each graph vary, because the individual matrix size is randomized.

The final observation is that from Figure 4.5 is that the set RAPL power cap has an effective impact in limiting the power draw, while gradually increasing the processing duration. Obviously, the shown system power draw is far higher that the given power caps for the respective graph, since the power cap limits only the CPU and the measured power draw includes the power draw of all components of the machine and power dissipation (e.g., from the power supply unit).

However, the power draw steps slightly different from expectations: Increasing the power cap from 8 W to 12 W rises the peak power draw by only 1.8 W, increasing the power cap further by 4 W up to 16 W in turn rises the peak power draw by 8.3 W—more than twice the amount, the next step in the power cap by 6 W to 22 W rises the peak draw by also 8.3 W. With the following step of 8 W to 30 W cap, the peak draw rises by 11.2 W, ind for the step of 10 W to 40 W cap, the peak

draw rises by 13.6 W. The last increase of the power cap by 20 W to 60 W cap, causes only the peak power draw to rise by 9.7 W. The run with out a power cap (dotted line) is similar to the graph of 60 W power cap, though it shows a slightly higher power draw of in average 0.3 W

The observation of the higher system power draw rises compared to the increase of the power cap can be explained with proportional increase of power dissipation and the overall rise of system activity due the effective higher activity of the CPU.

The anomaly at lower power caps can be explained by a base power draw of the CPU, which might be approximated with 10 W. A power cap further below this base power draw does not cause any further effect and the CPU keeps drawing the base power level. Therefore, the 12 W power cap is presumably just 1.8 W above this base draw, and thus increases the power draw by just this amount.

On the other hand, the anomaly at high power caps can be explained by reaching the maximum power, which the given implementation can cause on the present CPU. This effect becomes notably from the run without the power cap, which results in a system power draw of the peak by 88.2 W (ignoring the peak at the end of the peak) and the run with a cap of 40 W draws there already 81.4 W, which makes it reasonable that further increases of the power cap would cause reduced effects.

| RAPL cap | Duration | System avg. Power | Energy |
|---|---|---|---|
| 8 W | 88.71 s | 38.07 W | 3377 J |
| 12 W | 67.44 s | 38.70 W | 2610 J |
| 16 W | 67.42 s | 41.24 W | 2780 J |
| 22 W | 67.40 s | 42.89 W | 2891 J |
| 30 W | 67.39 s | 46.15 W | 3110 J |
| 40 W | 67.39 s | 50.05 W | 3373 J |
| 60 W | 67.39 s | 51.99 W | 3503 J |
| no cap | 67.40 s | 52.29 W | 3524 J |

**Table 4.4** – Results of OpenCL on Xeon executing m4096_i200 with different RAPL settings.

The presented results of Figure 4.5 are summarized in Table 4.4. The first column identifies summarized graph by the used RAPL power cap. The next two columns show the duration and average power draw of the system. The last column shows the required energy of the processing of the entire workload, which was the same in all listed cases.

The third column shows the already noticed reduction of the average power draw when the RAPL power cap is reduced. At the same time, reducing the power cap causes potentially an increase in the total processing duration. The reason that above a power cap of 22 W no further reduction can be noticed, is that the workload has a goal duration with inflicts delays, which compensate slight variation in actual processing time.

The energy values of Figure 4.5 show that there exists an optimal power cap (i.e., configuration) when the energy demand should be minimized. In this evaluation a power cap of 12 W required the minimal energy demand. If the too low power cap is chosen, the execution duration rises stronger then the power draw can be reduced, which causes higher energy demand. On the other hand, if the power cap is increased the rise in power draw becomes dominant over the reduction in the processing duration, which also causes higher energy demand.

However, should be highlighted again that the here evaluated workloads have a defined goal duration and faster processing than the goal duration causes the system to idle until the time is

eventually up. It reason behind implementing this goal duration is that in the context of application with interactive workloads, which are subject of this experiment, exists server which run a fix amount of time (e.g., the entire day), which is independent of the actual utilization of the system (e.g., amount of users sending request to the server). For example a server providing a service might run day and night to keep the service available, regardless that it might be used less intensive during the night time.

The evaluation of the effect of the configuration on the processing of workloads revealed two further results: First, the Quadro showed variations of the power draw when the GPU frequency is changed, however, the impact of the RAPL power cap on the CPU has a significantly higher effect on the power draw. The reason for this is, that on the one hand, the evaluated frequencies are rather close together leaving out the more extreme variations of the corner cases (circumstances limited the amount of frequencies to evaluate). On the other hand, during the execution of the kernel on the GPU runs one busy thread on the CPU presumably polling for the end of the kernel execution.

Second, the Intel HD is subject to RAPL power cap, because it is contained in the power domain controlled by RAPL. Unlike the Xeon, the Intel HD shows a maximum performance at a power cap of roughly 30 W. A lower power cap degrades the performance by the enforced the limit like shown for the CPU. However, with a power cap higher than 30 W or no power cap the performance degrades too, which seams to be unintended like a bug within the RAPL feature.

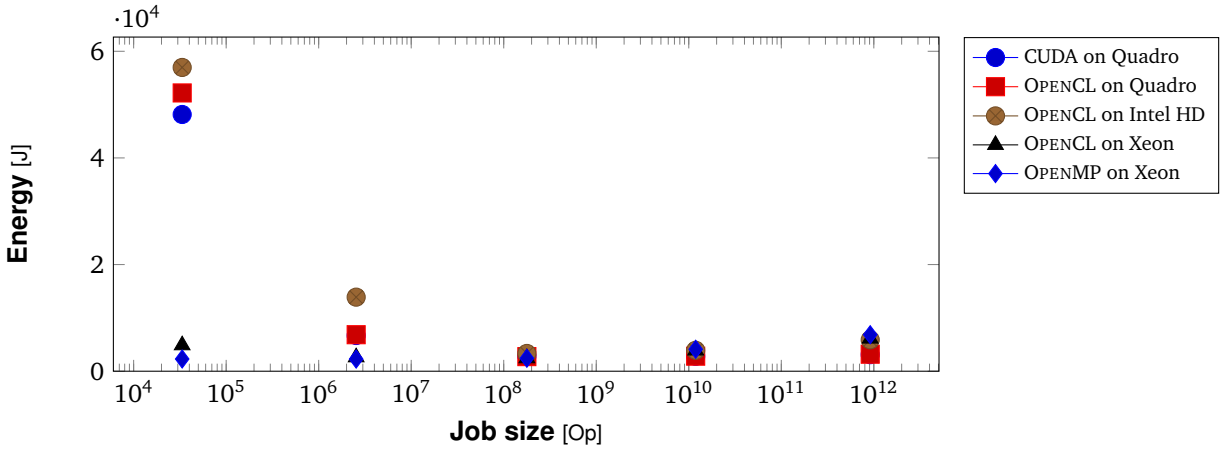



**Figure 4.6** – Energy usage by job size, shows the workload with suffix i0.

Figure 4.6 shows the energy demand of all five target executing the workloads with the suffix i0, each point shows the value of the configuration with the lowest energy demand for given combination of workload and target. The x-axis shows on a logarithmic scale the computational expense of a single job (i.e., estimated number of operation to decompose a single matrix) within the workload, which identifies the workload. The y-axis shows the energy demand for the execution of the specific workload on the specific target.

The graph shows the that the different targets execute the various workload requiring different amounts of energy. It can be seen that the Xeon executes the workload with the lowest job size significantly more efficient then the other target. In contrast, the Quadro processes the workload with the highest job size most efficiently.

Each job causes static execution overhead (i.e., calling the decomposition routine). This overhead is quite different for the different targets: the Xeon does a simple function call, while the GPUs targets the received job has to be forward to the GPU, which causes significant overhead.

This overhead becomes dominant for the GPUs smallest two job sizes, causing CUDA on Quadro and OpenCL on Intel HD to require up to 22 times and 27 times more energy, respectively, in processing the workloads compared to OpenMP on Xeon.

On the other hand, OpenMP on Xeon requires up to 4 times more energy than CUDA on Quadro for the workload with largest jobs. The reason for this is that the with such huge jobs the static execution overhead becomes negligible, a the more efficient parallel processing capacity of the GPUs becomes dominant.

Finally, the Figure 4.6 shows there is a job size ($1.77 \times 10^8$ operations, workload `m256_i0`), which requires relatively similar amount of energy of each target. At this job size, the least efficient target (OpenCL on Intel HD) requires only 1.77 times the energy of the most efficient target (OpenMP on Xeon). On average the if the worst target has 3.7 times the energy demand of the best target for the respective workload.

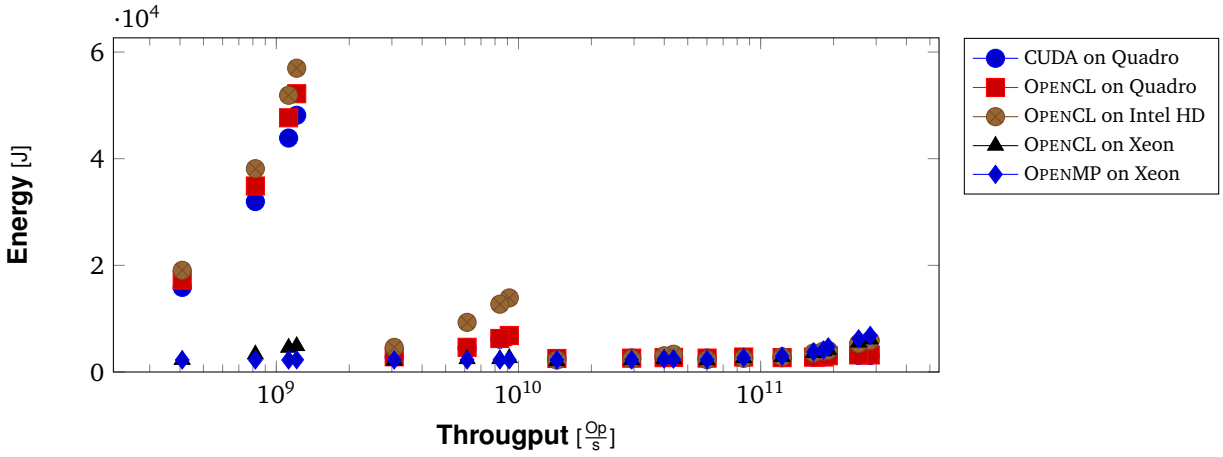



**Figure 4.7** – Energy demand by throughput.

Figure 4.7 shows the energy demand of all five target executing the all workloads, each point shows the value of the configuration with the lowest energy demand for given combination of workload and target. The x-axis shows on a logarithmic scale the computational throughput of the workload (see Table 4.3), which identifies the workload. The y-axis shows the energy demand for the execution of the specific workload on the specific target.

It is worth noticing that the shown throughput does not always pose a relation to level of utilization of the targets, because the different job sizes cause also varying levels of throughput, while keeping similar levels of utilization.

The figure shows essentially that the metric of computational throughput is insufficient for drawing conclusions regarding energy demand, because the energy demand of a single target has strong discontinuities along the throughput axis when the job size changes. For example OpenCL on Intel HD (the brown dots) shows monotonic energy demand when the throughput rises from $4 \times 10^8$ Op/s to $1.2 \times 10^9$ Op/s, as well from $3 \times 10^9$ Op/s to $9 \times 10^9$ Op/s, but the energy demand discontinuities from 62 kJ to 5.4 kJ between these two segments, because the job size is increased, which reduces the relative overhead. Therefore, the following two graphs each show only the results of the same job size, starting with the smallest job size.

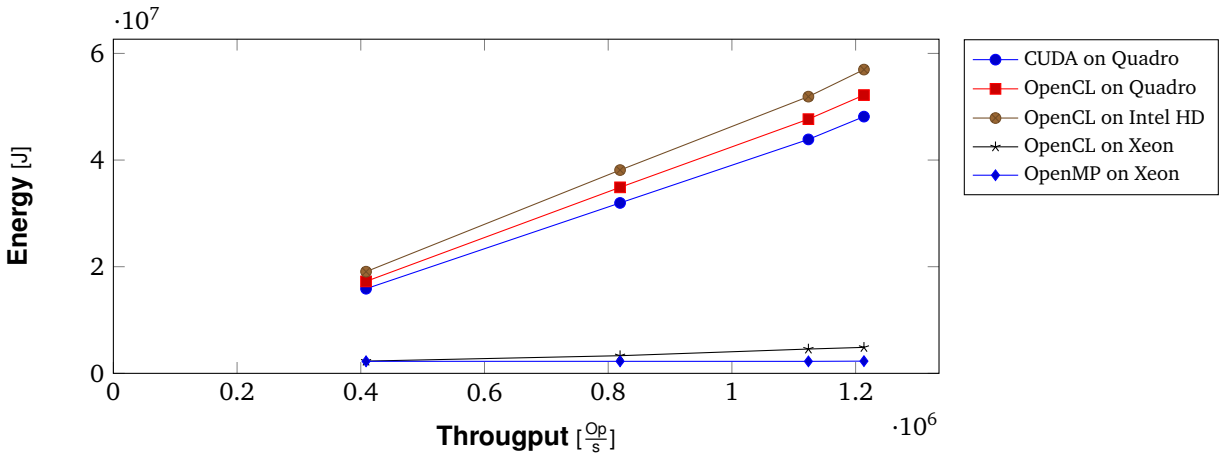



**Figure 4.8** – Energy demand by throughput for the smallest job size.

Figure 4.8 show the result of executing the workloads with the job size of 33.5 kOp: m16_i200, m16_i10, m16_i50, and m16_i0. The x-axis shows the computational throughput of the workload, and the y-axis the required energy demand of the execution of the workload on the specific target.

The graphs show independent of the target a nearly linear relation between the throughput and the energy demand. Additional, the figure shows that in this graph the OpenMP on Xeon is in all shown cases the most energy efficient target. However, Figure 4.6 that for other job size different target are the most energy efficient ones.

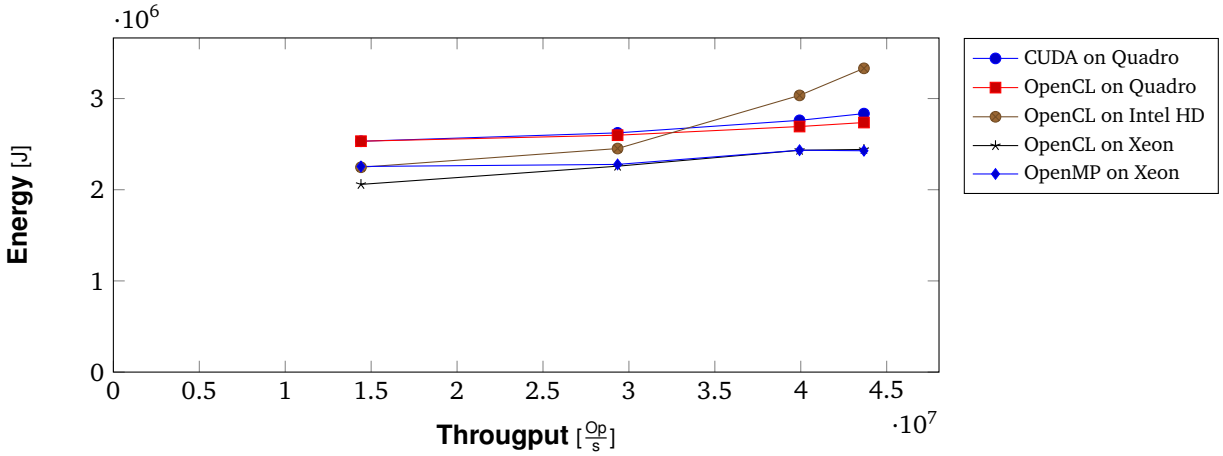



**Figure 4.9** – Energy demand by throughput for the smallest job size.

However, Figure 4.9 shows that there are job sizes which do not exhibit a linear energy–throughput relation. This figure has the same x-axis and y-axis as Figure 4.8 and shows the results of the workloads with a job size of 178 MOp: m256_i200, m256_i10, m256_i50, and m256_i0.

The presented Figures 4.6 to 4.9 showed that the depending on job size and computational throughput, different targets are the most energy efficient ones. Next the impact of the configuration (i.e., RAPL power cap and GPU frequency) on the energy efficiency will be discussed.

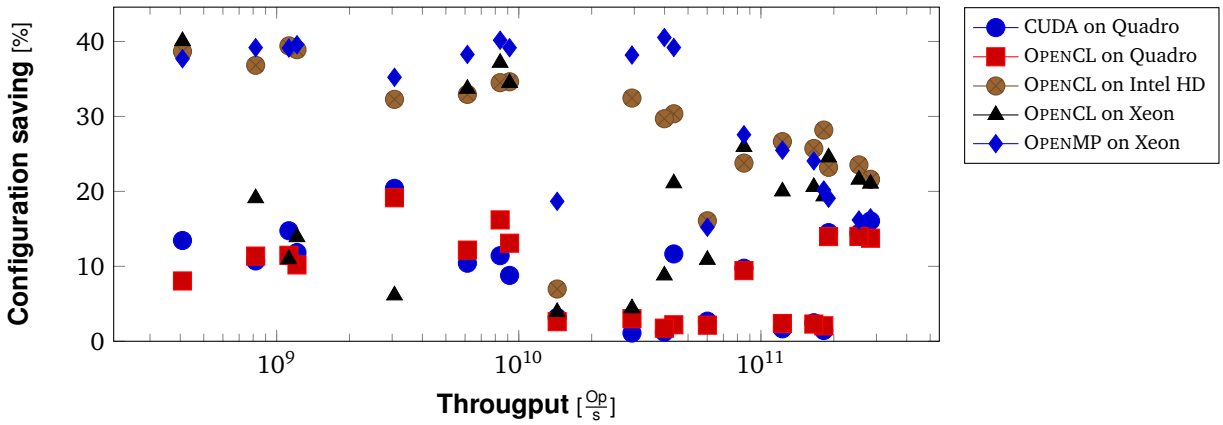



**Figure 4.10** – Energy reduction by applying the most energy efficient configuration compared to the default settings.

Figure 4.10 shows the energy saving of choosing the most energy efficient configuration compared to the energy demand when executed with the default configuration (e.g., no RAPL cap). The x-axis shows on a logarithmic scale the computational throughput of the workload (see Table 4.3), which identifies the workload. The y-axis shows the relative energy demand reduction for the specific workload and target.

The graph shows very strong fluctuations among the share of energy that can be saved. The highest saving was 40.5 % and the lowest 1.1 %, with an average saving of 19.4 %. The highest saving is achieved by OpenMP on Xeon with in average 30.5 %, whereas the lowest saving is achieved by OpenCL on Quadro with an average saving of 8.6 %,

| | `m16_i0` | `m256_i10` | `m4096_i50` |
|---|---|---|---|
| OpenMP on Xeon, RAPL cap | 8 W | 12 W | 22 W |
| OpenCL on Intel HD, RAPL cap | 12 W | 12 W | 22 W |
| CUDA on Quadro, RAPL cap | 12 W | 8 W | 8 W |
| CUDA on Quadro, GPU graphics clock | 1075 MHz | 1240 MHz | 810 MHz |

**Table 4.5** – The optimal power configuration for selected targets and workloads.

Table 4.5 shows for some targets and some workloads the respective most energy efficient configuration. The table shows that there is not a single configuration, which is always the most efficient one, instead each target requires for the different job sizes different configurations, to operate energy efficient.

In summary, the experiment presented in this section showed the energy demands of a set of heterogeneous hardware, and the impact of different configurations (e.g., RAPL power caps) on the energy efficiency of the execution on those components. It has been shown that choosing the right configuration reduces the energy demand by 19.4 % on average. It has been further shown that the least energy efficient target required on average 3.7 times the energy of the most energy efficient.

These two observation show that heterogeneous execution components can reduce the energy demand, which can be further reduced by using an optimal power cap or operation frequency. However, there is no target or configuration, which had always the most energy efficient operation.

This requires careful evaluation of the workloads and compute devices, as well as careful application of the found results. This becomes a tedious task if the number of hardware devices, size of the configuration space or number of jobs increases.

Consequently, using an automated tool such as a workload manager, which is aware of these parameters and can asses them automatically, will significantly reduce the work for the user and. Therefore, the thesis has introduced the concept of an workload manager Limo, which is aware of heterogeneity and the potential of exploiting this in order to increase the energy efficiency of the managed cluster (see Chapter 3). The following section compares Limo with an general-purpose workload manager, and thus shows its capabilities.

## 4.4 Workload Management Limo vs. SLURM

This section evaluates the effectiveness of the Limo prototype (see Section 3.3.1), by comparing it with the general-purpose workload manager SLURM. For the comparison, a one-hour workload composed of the NAS parallel benchmarks is executed by both workload managers. During the execution, varying power limits for the cluster are imposed and a real electricity price trace is projected into the evaluated hour, in order to calculate electricity cost generated by the two workload managers.

The cluster consists of four machines, two Xeon based machines ($Xeon_1$ and $Xeon_2$), one ODROID-C1+, and one ODROID-C2. The $Xeon_2$ has additionally one Intel HD and one Quadro. With this hardware setup yields four compute nodes (the four machines), however, one of them has additional GPUs,which yields six execution units for an heterogeneous aware workload manager.

SLURM, which is not heterogeneous aware, ignores therefore the two GPUs and distributes its job only among the four compute nodes. On the contrary, Limo considers the GPUs and distributes its jobs among all six execution units.

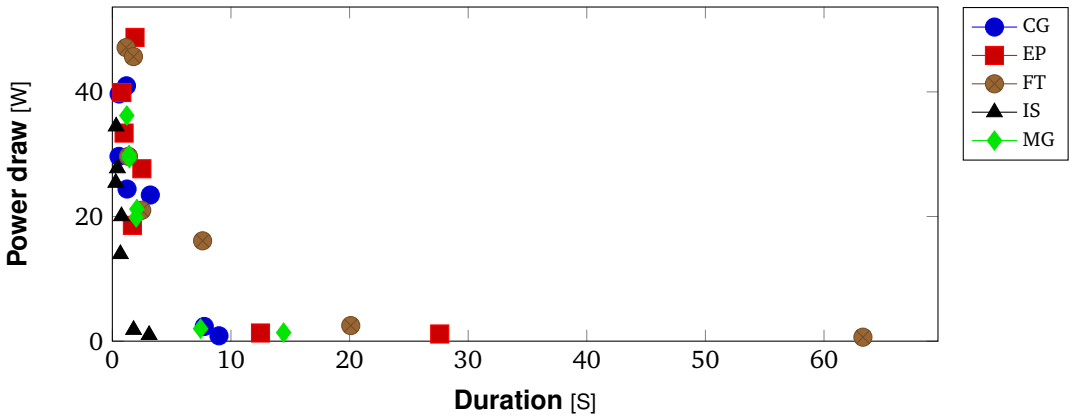



**Figure 4.11** – Power and time requirement of the different compute nodes by benchmark.

Figure 4.11 shows the power draw and duration of each benchmark by the different execution components available in the cluster. The x-axis of the figure shows the duration of the execution, and the y-axis shows the average power draw during the execution. Each benchmark has six points in the figure, one for each of the six execution units.

The figure shows that for each benchmark there are options to execute it with low power draw (dots along the x-axis) and alternatively to execute it with high power draw (dots along the y-axis).

Limo uses these alternatives once to implement the two operation modes (low power and high power, see Section 3.3.3) and adapt its operation to low power limits and high power limits, respectively.

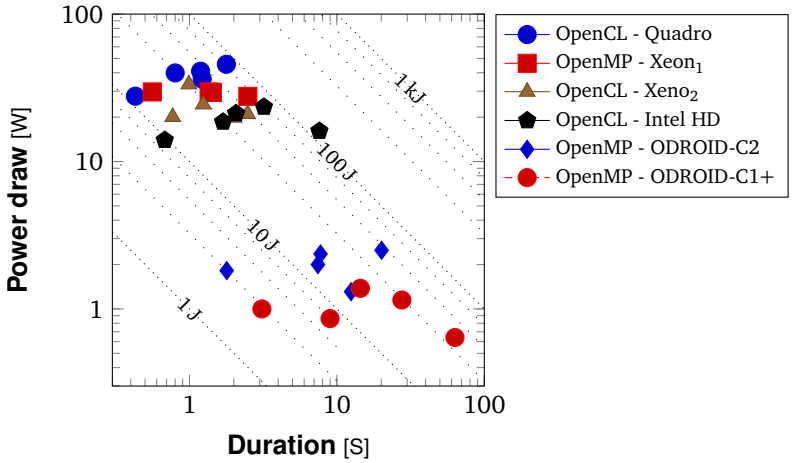



**Figure 4.12** – Power and time distribution of the NAS benchmarks by compute node in logarithmic scale on both axes. The dotted lines show the energy levels.

Figure 4.12 shows the same scatter plot as Figure 4.11, but the dots are distinguished by execution unit and both axes have logarithmic scaling. The x-axis of the figure shows the duration of the execution, and the y-axis shows the average power draw during the execution.

One speciality of this plot is that the energy demand of all dots along one dotted line (or any parallel line) is the same. Therefore, the figure shows also energy demand of each result for example the four dots enclosed by the 1 J and 10 J level are the results with the lowest energy demand of less than 10 J each.

Further, this figure reveals that the wide range of execution power draws for each benchmark originate from specific power draw level the different execution units exhibit. For example the Quadro contributes the results with the highest power draw for each benchmark and the ODROID-C1+ contributes the results with the lowest power draw, respectively.

| Time | Job count | Low limit | High limit |
|---|---|---|---|
| 0 min | 4 times | 39 W | 60 W |
| 10 min | 4 times | 60 W | 90 W |
| 20 min | 4 times | 39 W | 60 W |
| 30 min | 2 times | 65 W | 95 W |
| 40 min | 1 time | 100 W | 160 W |
| 50 min | 1 time | 80 W | 150 W |

**Table 4.6** – Specification of the workloads used in the evaluation Section 4.4.

The workload used in this experiment is designed to run of 60 min and is divided into six sections of 10 min duration. At the beginning of each section a new set of jobs is submitted to the workload manager and a new power constraint is set. Table 4.6 lists the six sections and there specification.

Each of the five NAS parallel benchmarks is used with the same frequency and all five benchmarks are submitted at once. Therefore, the the table lists only the submission count of the NPB set in the respective workload segment.

In the following, the results of executing this workload on the evaluation cluster once managed by SLURM and once by Limo are presented and analyzed regarding the obedience of the power limit and the generated electricity cost.

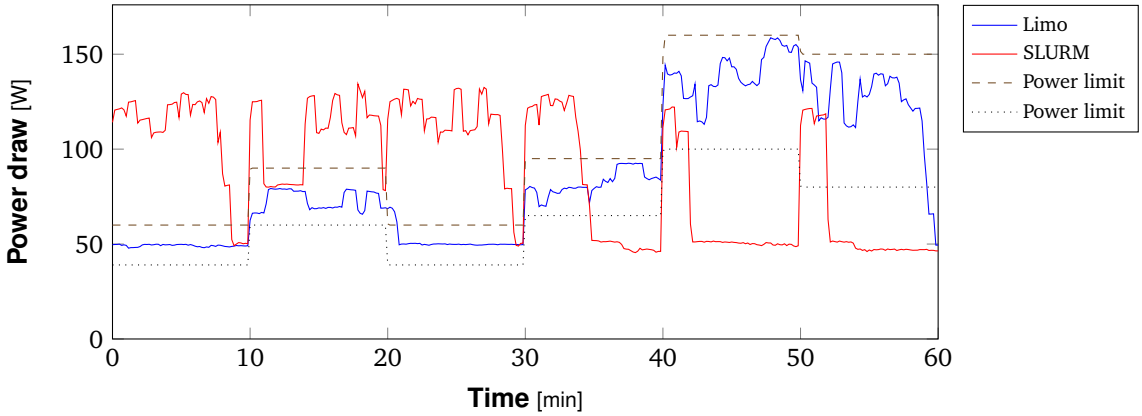



**Figure 4.13** – Power draw over time

**Figure 4.14** – Comparing Limo with SLURM regarding power constraints [Hön+18].

Figure 4.14 shows the total power draw of the evaluation cluster during execution of the workload of both workload managers, the dashed and dotted lines show the active power limits. The x-axis shows the execution time line, and the y-axis shows the power draw of the entire cluster.

The figure shows that SLURM executes the job immediately following their submission at the beginning of each section, regardless of the active power draw limits, which causes the cluster power draw to be only 16.9 % of the time in accordance with the power limits. The graph of SLURM demonstrates the power draw behavior of an workload manager that is not power aware, it exhibits significant and rapid power draw changes at on each, job submission and execution end of over 50 W. On the other hand, following the rapid execution times the cluster remains idle.

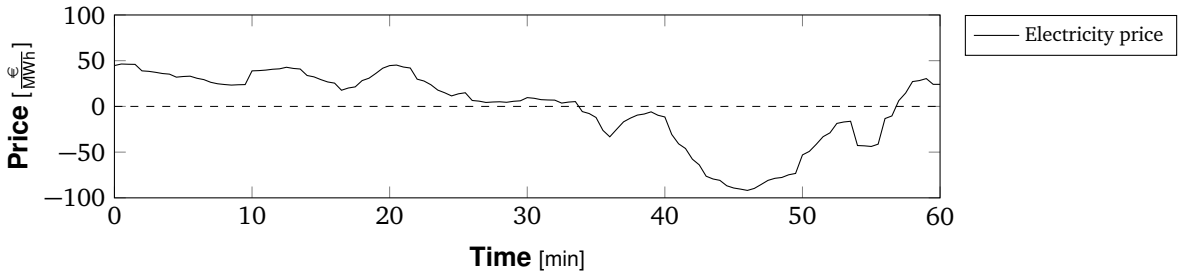
Opposing, Limo keeps the power draw of the cluster 97.5 % of the time within the given limits, the only exceptions are after 20 min and before 60 min. The first exception is caused by the reduction of the power limit, since Limo dose not implement job preemption, the running job exhibiting the allowed power draw of the previous section to continue until finished. The second exception before 60 min is caused by Limo running out of queued jobs.

Another difference between the power draw of SLURM and Limo is that Limo causes only power variations only within the allowed limits except for final power drop before 60 min.

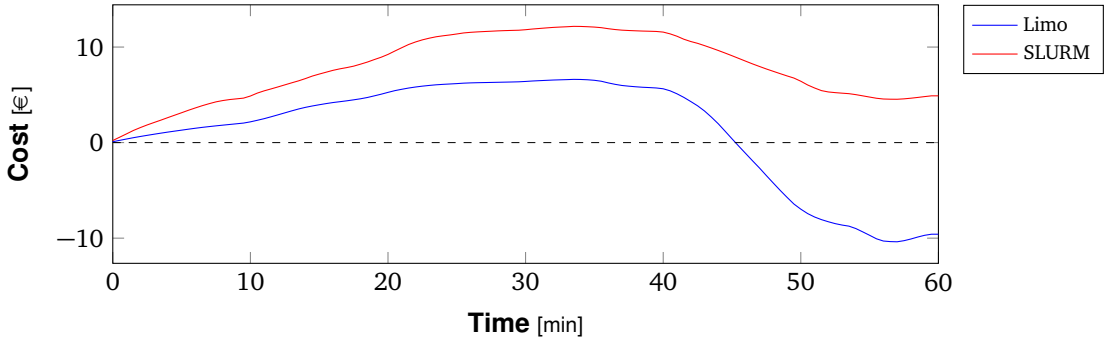
Figure 4.15 shows an example of caused electricity cost, the electricity price is take from the European energy stock exchange from a four day period between 26 and 29 October 2017 [Fra17], the power draw of Figure 4.14 is projected into this time frame resulting the shown cumulative cost.

The two x-axes in the figure show the time line of the execution, the y-axis of the upper graph shows the stock-exchange electricity price, and the y-axis of the lower graph shows the cumulative electricity cost of the cluster.

The price graph features negative electricity prices in the second half of the time frame. Limo shifts some of the plenty jobs submitted in the first half of the time frame into the second half, as




**(a)** Electricity price over time




**(b)** Cumulative cost

**Figure 4.15** – Comparing Limo with SLURM regarding electricity cost [Hön+18; Fra17].

shown in the previous figure (see Figure 4.14) by the power draw. Therefore, Limo profits stronger from the negative electricity prices than SLURM, at the same time, Limo saves cost in the first half of the time frame, where the electricity price is high.

As a result, Limo processes the given workload and gains a profit of 0.10€, while SLURM causes costs of 0.05€ in electricity cost, processing the same workload at in the same time frame with regards to the electricity pricing.

In summary, the experiment presented in this section showed that the evaluated hardware components can execute the benchmarks with a wide variety of power draw. Exploiting this variety, Limo can execute a given workload in accordance with various power limits and even take advantage of fluctuating electricity prices. In comparison with SLURM, Limo obeys the power limits in 97.5 % of the evaluated time, SLURM obeys it only 16.9 % of the time. At the same time, Limo generates a profit of 0.10€, whereas SLURM causes costs of 0.05€. This shows the effectiveness of the prototype regarding its power and price awareness.

## 4.5 Summary

This chapter showed first that a power cap can be formed for a Nvidia Quadro P2000 GPU using only frequency scaling. Following, it shows that heterogeneous hardware as well as enforcing appropriate power caps can reduce the energy demand by up to 73.0 % and 19.4 %, respectively, which is a significant increase of energy efficiency. Finally, this chapter showed the variety of execution power draws achieved available in an heterogeneous compute cluster, and that this can be exploited to

obey given power limits 97.5 % of the time compared to 16.9 % achieved by a workload manager which is not power aware, and by delaying execution of workloads Limo generates 0.10€ compared to a cost of 0.05€ caused by an workload manager which is not price aware.

# 5 CONCLUSION

This chapter finally summarizes the results of this thesis and gives a prospect for future work. Centrally, this thesis regards power-grid stability, energy-demand reduction, and electricity-price considerations in the context of cluster computing. Problems in this domain are rapid power draw changes, high energy demand, and expensive electricity costs of the clusters.

Therefore, the concept of the workload manager Limo was presented, in order to propose a system that solves these problems. Limo obeys a given power draw target zone, which reduces its power draw changes. It improves the energy efficiency of the cluster, which reduces its energy demand, and Limo monitors the electricity price to priories job execution during times of lower prices, which reduces its operating costs. Limo achieves this goals by exploiting heterogeneity within the compute cluster,installing software power caps on the compute nodes and selectively delaying job execution.

Further, the implementation of a utility program for Limo is described, which generates job execution profile databases, and a prototype of the Limo concept is explained, which implementing power and price awareness and obeys given power limits while reducing operating costs. The first experiment conducted for this thesis showed that purposeful specification of GPU clock frequency, resulted in effective power capping of the evaluated GPU.

The analysis of the results gathered by the utility program reveal a potential of reducing the energy demand by 73.0 % and 19.4 % by using different execution hardware and installing the optimal power cap, respectively. The comparison of the Limo prototype with a general-purpose workload manager that is neither power nor price aware showed that Limo obeys the power limits 97.5 % of the time, compared to 16.9 % achieved by the other workload manager. A projection of the same results and combination with an actual price curve showed, the the compared workload manager accumulated 0.05€ in electricity cost, while Limo generated a profit from electricity pricing, which turned temporarily negative, of 0.10€.

In summary, the thesis showed that heterogeneity in a computer cluster exhibits characteristics, which can be effectively exploited in order to increase energy efficiency, comply with power limits, and cut operating costs. The first effect of increase energy efficiency have been indirectly shown by analyzing the execution results of one benchmark on a variety of heterogeneous hardware, while the latter two effects have been demonstrated by an prototype compared with a widely used workload manager.

However, the evaluated workload manager was only a prototype which did not efficiently implemented the effect of increasing the energy efficiency, and dose not trade off the obedience of power limits against the opportunities (e.g., low electricity price) or risks (e.g., upcoming job deadlines). Thus, extending the prototype by an multi-goal optimization heuristic would complete its feature set. Additional features that would improve Limo, are using job preemption like a preemptive

scheduler, to interrupt jobs when power limits decline, and changing the power caps during the execution of a job, in order to dynamically adapt the change circumstances (e.g., in power limit or electricity price).

# LIST OF ACRONYMS

**ACPI** Advanced Configuration and Power Interface

**API** Application Programming Interface

**APM** Advanced Power Management

**BLOB** Binary Large OBject

**CPU** Central Processing Unit

**CUDA** Compute Unified Device Architecture

**DFS** Dynamic Frequency Scaling

**DVFS** Dynamic Voltage and Frequency Scaling

**DVS** Dynamic Voltage Scaling

**ELF** Executable and Linkable Format

**EOF** End of File

**GPU** Graphics Processing Unit

**HPC** High Performance Computing

**ID** Identifier

**I/O** Input/Output

**IP** Internet Protocol

**ISO** International Organization for Standardization

**LUD** Lower Upper Decomposition

**NAS** NASA Advanced Supercomputing Division

**NASA** National Aeronautics and Space Administration

**NPB** NAS Parallel Benchmarks

**OpenCL** Open Computing Language

**OpenMP** Open Multi-Processing

**OS** Operating System

**OSI** Open Systems Interconnection model

**OSS** Open-Source Software

**PCIE** Peripheral Component Interconnect Express

**POSIX** Portable Operating System Interface

**QOS** Quality of Service

**RAPL** Running Average Power Limit

**SBC** Single-Board Computer

**SNU** Seoul National University

**TCP** Transmission Control Protocol

**USB** Universal Serial Bus

# LIST OF FIGURES


2.1 Comparing Limo with SLURM regarding electricity cost. . . . . 8

3.1 Overview of SLURM's components. . . . . 14
3.2 Overview of Limo's components. . . . . 15
3.3 Dynamic power limits forming penalty zones. . . . . 16
3.4 Scheme of Limo's dispatch algorithm. . . . . 17
3.5 The components of the profile generator prototype. . . . . 19
3.6 Profile generator communication protocol. . . . . 22
3.7 Workload manager components. . . . . 28
3.8 Workload-manager communication protocol. . . . . 32

4.1 The cluster setup of the evaluation. . . . . 38
4.2 Impact of all configurable GPU frequencies on a Nvidia Quadro P2000 execution LUD of a 16384 × 16384 Matrix—Execution Duration . . . . . 42
4.3 Impact of all configurable GPU frequencies on a Nvidia Quadro P2000 execution LUD of a 16384 × 16384 Matrix—Average Power Draw . . . . . 43
4.4 Impact of all configurable GPU frequencies on a Nvidia Quadro P2000 execution LUD of a 16384 × 16384 Matrix—Benchmark efficiency . . . . . 44
4.5 The effect of DVFS on CPU via RAPL. Shows OpenCL on Xeon executing m4096_i200. 46
4.6 Energy usage by job size, shows the workload with suffix i0. . . . . 48
4.7 Energy demand by throughput. . . . . 49
4.8 Energy demand by throughput for the smallest job size. . . . . 50
4.9 Energy demand by throughput for the smallest job size. . . . . 50
4.10 Energy reduction by applying the most energy efficient configuration compared to the default settings. . . . . 51
4.11 Power and time requirement of the different compute nodes by benchmark. . . . . 52
4.12 Power and time requirement of the NAS benchmarks by compute node. . . . . 53
4.13 Power draw over time . . . . . 54
4.14 Comparing Limo with SLURM regarding power constraints. . . . . 54
4.15 Comparing Limo with SLURM regarding electricity cost. . . . . 55

# LIST OF TABLES

2.1 Comparison of different workload managers. . . . . 10

3.1 Protocol status codes . . . . . 21

4.1 Overview of the evaluated devices with system- and component-scope heterogeneity. 37
4.2 Description of the benchmarks used in the workload manager evaluation. . . . . 40
4.3 Specification of the workloads used in the evaluation Section 4.3. . . . . 45
4.4 Results of OpenCL on Xeon executing m4096_i200 with different RAPL settings. . . . 47
4.5 The optimal power configuration for selected targets and workloads. . . . . 51
4.6 Specification of the workloads used in the evaluation Section 4.4. . . . . 53

# LIST OF LISTINGS

3.1 Connection protocol data . . . . . 20
3.2 Application interface . . . . . 26
3.3 Example implementation of the application interface . . . . . 27
3.4 Weak application declaration . . . . . 27
3.5 Workload manager protocol data—`Type` and `Block` . . . . . 29
3.6 Workload manager protocol data—`NewSlave` . . . . . 29
3.7 Workload manager protocol data—`SendExec` . . . . . 30
3.8 Workload manager protocol data—`DoExec` . . . . . 30
3.9 Workload manager protocol data—`JobAck` . . . . . 30
3.10 Workload manager protocol data—`StreamData` . . . . . 30
3.11 Workload manager protocol data—`TypeConf` . . . . . 31

# LIST OF GRAMMARS

3.1 Workload file format . . . . . 23

# LIST OF ALGORITHMS

3.1 Generator algorithm . . . . . 24
3.2 Executor algorithm . . . . . 25
3.3 Job dispatch algorithm of the Limo prototype . . . . . 33

# REFERENCES


[Bai+91] David H Bailey et al. “The NAS parallel benchmarks.” In: *The International Journal of Supercomputing Applications* 5.3 (1991), pp. 63–73.

[Bar05] Luiz André Barroso. “The price of performance.” In: *Queue* 3.7 (2005), pp. 48–53.

[BH07] L. A. Barroso and U. Hölzle. “The case for energy-proportional computing.” In: *Computer* 40.12 (Dec. 2007), pp. 33–37.

[Che+09] Shuai Che et al. “Rodinia: A benchmark suite for heterogeneous computing.” In: *2009 IEEE International Symposium on Workload Characterization (IISWC ’09)*. Ieee. 2009, pp. 44–54.

[Dun16] Jill Dunbar, ed. *NAS parallel benchmarks*. NASA. 2016. URL: `https://www.nas.nasa.gov/publications/npb.html` (visited on 06/20/2018).

[FK97] Michael Franz and Thomas Kistler. “Slim binaries.” In: *Commun. ACM* 40.12 (Dec. 1997), pp. 87–94.

[Gor15] Ryan C Gordon. *FatELF: Universal Binaries for Linux*. 2015. URL: `http://icculus.org/fatelf/` (visited on 05/21/2018).

[Har+09] Stavros Harizopoulos, Mehul Shah, Justin Meza, and Parthasarathy Ranganathan. “Energy efficiency: The new holy grail of data management systems research.” In: *Fourth Biennial Conference on Innovative Data Systems Research (CIDR ’09)*. CIDR 2009. Asilomar, CA, USA, Jan. 2009.

[Hem10] Scott Hemmert. “Green HPC: From nice to necessity.” In: *Computing in Science Engineering* 12.6 (Nov. 2010), pp. 8–10.

[HL04] Karl Hillesland and Anselmo Lastra. “GPU floating-point paranoia.” In: *Proceedings of GP2* 318 (2004).

[Hön+18] Timo Hönig, Christopher Eibel, Adam Wagenhäuser, Maximilian Wagner, and Wolfgang Schröder-Preikschat. “How to make profit: Exploiting fluctuating electricity prices with Albatross, a runtime system for heterogeneous HPC clusters.” In: *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS ’18)*. ACM. Tempe, AZ, USA, 2018, pp. 1–9.

[IA09] A. Ipakchi and F. Albuyeh. “Grid of the future.” In: *IEEE Power and Energy Magazine* 7.2 (Mar. 2009), pp. 52–62.

[Kim+12] Jungwon Kim et al. “SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters.” In: *Proceedings of the 26th ACM international conference on Supercomputing (ICS ’12)*. ACM. 2012, pp. 341–352.

[Lin+09] B. Lin, A. Mallik, P. Dinda, G. Memik, and R. Dick. "User- and process-driven dynamic voltage and frequency scaling." In: *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '09)*. Apr. 2009, pp. 11–22.

[MGW09] David Meisner, Brian T Gold, and Thomas F Wenisch. "PowerNap: Eliminating server idle power." In: *ACM Sigplan Notices*. Vol. 44. 3. ACM. 2009, pp. 205–216.

[MWC17] Xinxin Mei, Qiang Wang, and Xiaowen Chu. "A survey and measurement study of GPU DVFS on energy conservation." In: *Digital Communications and Networks* 3.2 (May 2017), pp. 89–100.

[Qia16] Depei Qian. *High performance computing: A brief review and prospects*. 2016.

[Ree17] Stanley Reed. "Power prices go negative in Germany, a positive for consumers." In: *The New York Times* 167 (57823 Dec. 25, 2017), B3.

[Seo+13] Sangmin Seo et al. *SNU NPB suite*. Seoul National University. Feb. 26, 2013. URL: http://aces.snu.ac.kr/software/snu-npb/ (visited on 04/13/2018).

[Tsa+05] Dan Tsafrir, Yoav Etsion, Dror G Feitelson, and Scott Kirkpatrick. "System noise, OS clock ticks, and fine-grained parallel applications." In: *Proceedings of the 19th annual international conference on Supercomputing (ICS '05)*. ACM. 2005, pp. 303–312.

[Wag17] Maximilian Wagner. "Improving the energy efficiency of a many-node heterogeneous computing system utilizing application-induced energy claims." Bachelor's Thesis. University of Erlangen, Dept. of Computer Science, May 2017.

[Wan+17] Qiang Wang, Pengfei Xu, Yatao Zhang, and Xiaowen Chu. "EPPMiner: An extended benchmark suite for energy, power and performance characterization of heterogeneous architecture." In: *Proceedings of the Eighth International Conference on Future Energy Systems (e-Energy '17)*. ACM. 2017, pp. 23–33.

[WCS15] Ke Wang, Shuai Che, and Kevin Skadron. *Rodinia suite*. University of Virginia. Dec. 12, 2015. URL: https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators (visited on 12/22/2017).

[WS15] Harry Wirth and Karin Schneider. "Recent facts about photovoltaics in Germany." In: *Report from Fraunhofer Institute for Solar Energy Systems* (2015).

[YJG03] Andy B Yoo, Morris A Jette, and Mark Grondona. "Slurm: Simple linux utility for resource management." In: *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '03)*. Springer. 2003, pp. 44–60.

[Yu+17] Hangchen Yu, Ke Wang, Shuai Che, and Kevin Skadron. *GPU Rodinia*. Github. Oct. 23, 2017. URL: https://github.com/yuhc/gpu-rodinia (visited on 04/13/2018).

[ARM13] ARM Limited. *big.LITTLE Technology: The Future of Mobile*. 2013. URL: https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf.

[Ada18] Adaptive Computing Inc. *Moab HPC suite solution architecture*. 2018. URL: http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/basic-edition-solution-architecture (visited on 06/20/2018).

[Cha17] Chauvin Arnoux Group. *Current clamp for AC/DC current*. Jan. 2017. URL: http://www.chauvin-arnoux.com/sites/default/files/D00YWK68_5.PDF (visited on 06/20/2018).

[Fra17] Fraunhofer ISE. *Electricity production and spot prices in Germany*. Oct. 2017. URL: www.energy-charts.de/price.htm (visited on 06/20/2018).

[IEE93] IEEE Portable Applications Standards Committee and others. *IEEE Std 1003.1 b-1993, Real Time Extensions*. IEEE, 1993.

[ISO99] ISO/IEC JTC 1/SC 22 Joint Technical Committee portability subcommittee. *ISO/IEC 9899:1999, Programming languages – C*. ISO/IEC, 1999.

[ITU08] ITU Telecommunication standardization sector. *ITU-T E.800, Definitions of terms related to quality of service*. ITU, Sept. 23, 2008.

[Int18] Intel Corporation. *Intel 64 and IA-32 architectures software developer's manual, volume 3B: system programming guide*. May 2018.

[Khr15] Khronos OpenCL Working Group. *The OpenCL Specification*. Ed. by Lee Howes and Aaftab Munshi. Version 2.0. Khronos Group Inc. July 21, 2015. URL: https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf (visited on 06/20/2018).

[Lin17] Linux man-pages project. *feature_test_macros(7) Linux Programmer's Manual*. 4.13. Sept. 15, 2017.

[Mic15] Microchip Technology Inc. *MCP39F511. Power-Monitoring IC with Calculation and Energy Accumulation*. 2015. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/20005393B.pdf (visited on 06/20/2018).

[Nvi16] Nvidia Corporation. *NVIDIA System Management Interface program*. July 26, 2016. URL: http://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf (visited on 06/20/2018).

[Nvi18] Nvidia Corporation. *Cuda C programming guide*. PG-02829-001_v9.2. May 2018. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (visited on 06/20/2018).

[Ope13] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. Version 4.0. July 2013. URL: https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf (visited on 06/20/2019).

[Pro17a] Prometeus GmbH. *Green 500 List*. Nov. 2017. URL: https://www.top500.org/green500/list/2017/11/ (visited on 04/19/2018).

[Pro17b] Prometeus GmbH. *Top 500 List*. Nov. 2017. URL: https://www.top500.org/list/2017/11/ (visited on 04/19/2018).

[Ste18] StemLabs, ed. *Red Pitaya Stemlab board*. 2018. URL: https://www.redpitaya.com/f130/STEMlab-board (visited on 06/20/2018).

[Uni18] Univa Corporation. *Univa Grid Engine*. 2018. URL: http://www.univa.com/resources/files/gridengine.pdf (visited on 06/20/2018).

[Wik11] Wikipedia contributors. *Variation of utility frequency over 48 hours for some european and asian countries*. File: Variation of utility frequency.svg. Wikimedia Commons. Feb. 15, 2011. URL: https://en.wikipedia.org/wiki/File:Variation_of_utility_frequency.svg (visited on 06/20/2018).