A Unified Infrastructure for Monitoring and Tuning the Energy Efficiency of HPC Applications

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Robert Schöne
geboren am 26. März 1981in Belzig

Gutachter: Prof. Dr. rer. nat. Wolfgang E. Nagel, Technische Universität Dresden
Prof. Dr. rer. nat. Thomas Ludwig, Universität Hamburg

Tag der Einreichung: 15. März 2017
Tag der Verteidigung: 19. September 2017

Dresden, 6. November, 2017

Abstract

High Performance Computing (HPC) has become an indispensable tool for the scientific community to perform simulations on models whose complexity would exceed the limits of a standard computer. An unfortunate trend concerning HPC systems is that their power consumption under high-demanding workloads increases. To counter this trend, hardware vendors have implemented power saving mechanisms in recent years, which has increased the variability in power demands of single nodes. These capabilities provide an opportunity to increase the energy efficiency of HPC applications. To utilize these hardware power saving mechanisms efficiently, their overhead must be analyzed. Furthermore, applications have to be examined for performance and energy efficiency issues, which can give hints for optimizations. This requires an infrastructure that is able to capture both, performance and power consumption information concurrently. The mechanisms that such an infrastructure would inherently support could further be used to implement a tool that is able to do both, measuring and tuning of energy efficiency.

This thesis targets all steps in this process by making the following contributions: First, I provide a broad overview on different related fields. I list common performance measurement tools, power measurement infrastructures, hardware power saving capabilities, and tuning tools. Second, I lay out a model that can be used to define and describe energy efficiency tuning on program region scale. This model includes hardware and software dependent parameters. Hardware parameters include the runtime overhead and delay for switching power saving mechanisms as well as a contemplation of their scopes and the possible influence on application performance. Thus, in a third step, I present methods to evaluate common power saving mechanisms and list findings for different x86 processors. Software parameters include their performance and power consumption characteristics as well as the influence of power-saving mechanisms on these. To capture software parameters, an infrastructure for measuring performance and power consumption is necessary. With minor additions, the same infrastructure can later be used to tune software and hardware parameters. Thus, I lay out the structure for such an infrastructure and describe common components that are required for measuring and tuning. Based on that, I implement adequate interfaces that extend the functionality of contemporary performance measurement tools. Furthermore, I use these interfaces to conflate performance and power measurements and further process the gathered information for tuning. I conclude this work by demonstrating that the infrastructure can be used to manipulate power-saving mechanisms of contemporary x86 processors and increase the energy efficiency of HPC applications.

Contents

1 Introduction and Motivation

> *once you get the beginning right, the ending almost writes itself*
> **Speaker for the Dead** (Introduction) by Orson Scott Card

In the last decades, computer simulations have become an indispensable tool to research the behavior of complex systems [Vet15]. They are used to solve real-life problems based on mathematical models. Therefore, they can be applied if the simulated event is not suitable for real-life testing (e.g., earthquakes) or the tested objects do not yet exist. For example, to simulate the effects of the air flow around a wing of an airplane, a computational fluid dynamics (CFD) simulation can use the adaptive grid method [FP02, Section 11.4]. However, conventional computers do not provide the necessary resources to hold all the data needed for such a simulation. Furthermore, their computing resources are limited. Hence, a fine-grained simulation can take a significant amount of time before the result is available. While this can be accepted in some cases, others have real-time constraints. A weather forecast for a specific day, for example, should be available prior to that day. Contemporary High Performance Computing (HPC) systems provide millions of processor cores and multiple petabytes of main memory. This enables researchers to simulate their problems at a fine-grained resolution within an appropriate time span.
The performance of such HPC systems is steadily increasing over time as shown in Figure 1.1. The used data is gathered from the Top500 list [Meu08], which publishes an overview on the fastest 500 HPC systems twice a year. It visualizes the floating point performance of the highest ranked and the lowest ranked system as well as the aggregated performance of all systems in the list. Power consumption and computing efficiency of the systems since 2007 are noted in the Green500 list [SHcF06, FS09]. Figure 1.2 shows the power consumption of the highest and lowest ranked system in the Top500 list and the aggregated power consumption of all systems. The figures indicate a trend to a higher performance as well as to a higher power consumption. However, both lists rely only on performance and power results measured for the LINPACK benchmark, which utilizes the computing resources very efficiently from a performance point of view. This results in a high power consumption [HOMS13]. However, most HPC applications do not utilize the hardware to the same extend. Therefore, their power consumption is typically lower [HSM+10].

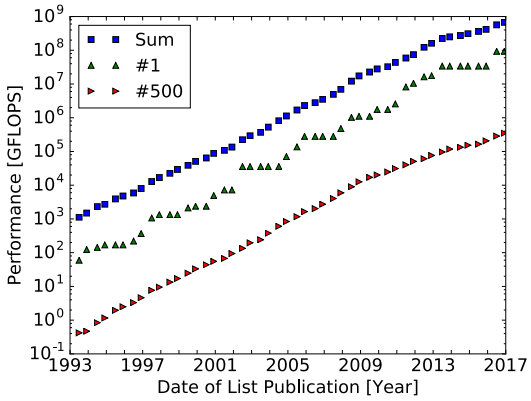


Figure 1.1: Development of HPC system performance, based on results from top500.org. The plot visualizes the performance of the highest ranked (#1) and lowest ranked (#500) system as well as the aggregated performance (Sum) of all listed systems.

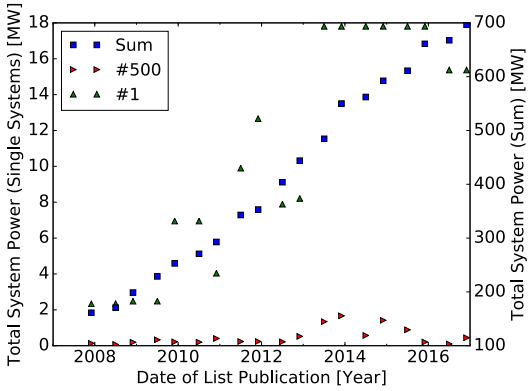
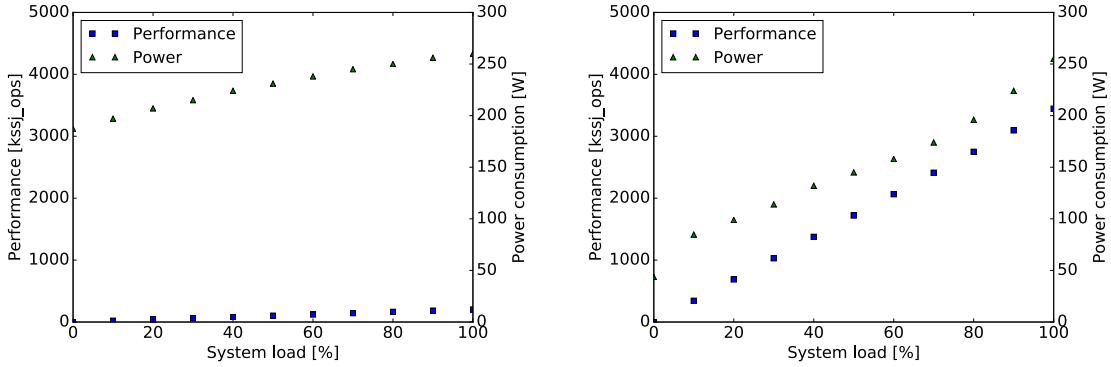


Figure 1.2: Development of HPC system power consumption, based on results from top500.org/green500/. The plot shows power consumption for the systems given in Figure 1.1, beginning 11/2007.



(a) Results of a 2007 dual socket Fujitsu Siemens system

(b) Results of a 2016 dual socket Fujitsu system

Figure 1.3: Performance and power consumption of two dual socket Intel x86 systems for the SPECpower_ssj benchmark depending on the system load.

In recent years, hardware vendors implemented power saving techniques that are utilized by operating systems to lower the power draw of certain components [acp16]. This results in a higher variability in the power consumption of single nodes[1]. The SPECpower_ssj 2008 benchmark [Lan09] measures this variability by issuing different levels of load to the tested system. In Figure 1.3, I compare the first tested dual-socket server by the vendor Fujitsu[2] with a more recent one. The older system[3] from 2007 has a narrow power range, where 187 Watt are drawn on an idle system and 260 Watt under full load. The newer system[4] from 2016 has almost the same maximal power consumption (255 Watt) but a significantly decreased idle power of 43.6 W. The power consumption under partial load is reduced as well. Here, the operating system uses the power saving techniques that are supported by the hardware to apply a more energy-efficient configuration. This results in an increased energy efficiency. These mechanisms can also be used to increase the energy efficiency when the system is under full load [GHBDL09]. Therefore, a framework that enables users and administrators of HPC systems to apply an energy-efficient configuration seems desirable. This would help to reduce electricity costs and the carbon footprint.
An energy efficiency tuning framework targeted at HPC needs to fulfill several requirements. First, it must be able to *access existing power measurement infrastructures* to find hot-spots and verify whether an applied tuning really increased the energy efficiency. Furthermore, it must be operational on *common HPC platforms*, support the *typical programming languages* and *paradigms*, and *access available power saving methods*. When these demands are met, the energy efficiency of a given HPC application can be analyzed and tuned, as depicted in Figure 1.4.

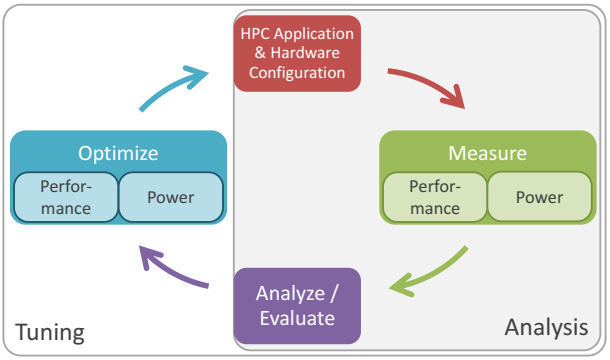
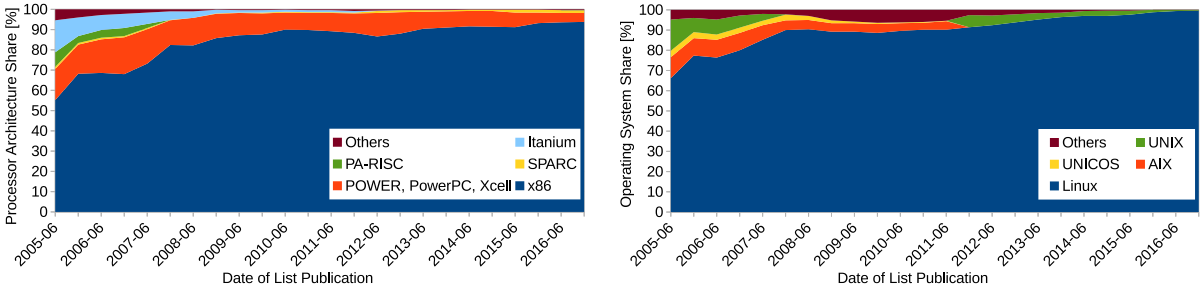


Figure 1.4: Performance and energy efficiency analysis and tuning cycle. Such a tuning infrastructure needs mechanisms to measure the performance of existing HPC workloads, access power measurement capabilities, and access power saving and performance tuning mechanisms.

[1] A node is a single self contained compute unit, which runs one operating system instance.
[2] resp., Fujitsu Siemens
[3] http://www.spec.org/power_ssj2008/results/res2007q4/power_ssj2008-20071128-00001.html
[4] http://www.spec.org/power_ssj2008/results/res2016q2/power_ssj2008-20160412-00722.html



(a) Instruction Set Architectures (ISAs)

(b) Operating Systems

Figure 1.5: System share of processor architectures and operating systems in Top500 HPC list.

The availability of power measurement infrastructures on HPC systems depends on the support from vendors [HIS+14, FCGS14, KFH+14] and the capabilities of the used processors [HIS+13, FAWR+11]. Due to the variety of existing power meters, a measurement framework should provide an interface that is able to cope with the different sources. Fortunately, most HPC systems use the same processors with the same instruction set architecture (ISA) and operating system (OS), as depicted in Figure 1.5a and Figure 1.5b, respectively. In the most recent list, 93.6 % of all systems used x86 based processors and 99.6 % used a Linux based OS. The remaining systems mostly use POWER based processors, which have a share of 4.4 %. However, a framework that relies on Linux and x86 inherently supports the predominant majority of the existing HPC systems.

In order to determine dominant programming languages for HPC software, I analyzed benchmark suites that target HPC platforms. The Standard Performance Evaluation Corporation (SPEC) develops benchmarks for different use cases, for example to evaluate the performance of Central Processing Units (CPUs), graphic workstations, or web servers. For HPC systems, the SPEC high performance group provides three different benchmark suites: SPEC ACCEL [JBC+15], which targets accelerator devices like general purpose graphics processing units (GPGPUs), SPEC MPI2007 [MvWL+09] which makes use of the Message Passing Interface (MPI), and SPEC OMP2012 [MBB+12], which uses the Open Multi-Processing (OpenMP) paradigm for a thread parallel execution of workloads. All benchmarks within the three suites represent scientific workloads used in HPC. One alternative, which uses real applications for evaluating the performance of highly parallel computers, is the CORAL benchmark suite [WT15]. This suite has been defined by three US national laboratories, namely Oak Ridge, Argonne, and Livermore, to ensure that offered HPC systems are capable of running typical workloads with a high throughput. Another benchmark suite that tests parallel systems are the NASA Advanced Supercomputing (NAS) Division's NAS Parallel Benchmarks (NPBs) [BBB+94]. In Figure 1.6, I visualize which programming languages are used in the individual benchmark suites. While the NPBs mostly rely on Fortran, the

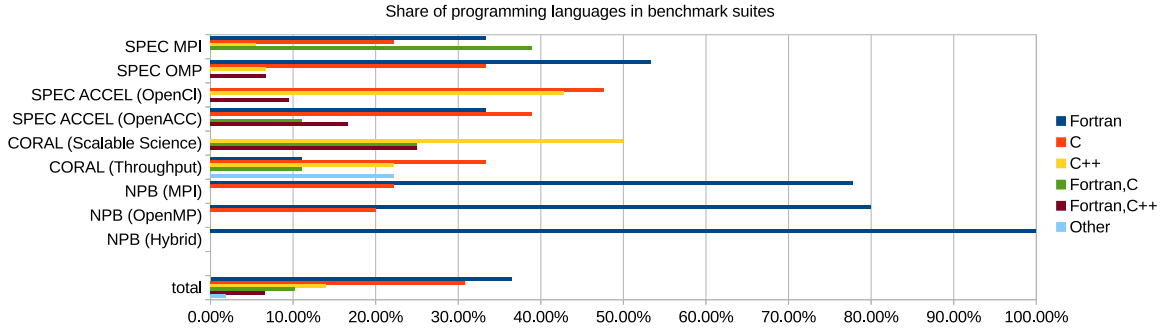


Figure 1.6: Used programming languages in different HPC benchmarking suites. The total distribution is calculated by creating a superset which contains every benchmark of the analyzed suites. Other languages include Python scripts added to mixes of C, C++, and Fortran.

CORAL benchmarks mostly use C/C++. The SPEC benchmarks use both to an equal extend, with the exception of the SPEC ACCEL OpenCL part. However, with the exception of Python, which is used by CORAL throughput benchmarks, no other programming language is used. Hence, an energy efficiency optimization framework should be able to interface C, C++, and Fortran. With the exception of SPEC ACCEL, all benchmark suites rely on MPI and OpenMP for process and thread parallelization, respectively. Therefore, the targeted infrastructure should also be able to interface these parallelization paradigms. Support for other libraries, frameworks, and accelerators is preferable.
In this thesis, I describe how I designed an infrastructure that targets energy efficiency measurement and tuning. It uses existing performance monitoring tools that already support the targeted operating systems, processor architectures, programming languages, and parallelization paradigms. The proposed infrastructure enhances them by interfaces that can be used to incorporate power information into the existing infrastructure and to tune the energy efficiency of the analyzed system.
This thesis is structured as follows: I introduce the used nomenclature and existing technology in Chapter 2. This includes an overview on contemporary performance and power measurement infrastructures, power saving mechanisms, and energy efficiency tuning types. In Chapter 3, I introduce a model that can be used to determine and describe energy-efficient configurations on a code region level. This model is based on hardware and software parameters that describe the implications of power saving mechanisms on performance and power consumption. The hardware parameters are detailed in Chapter 4. There, I list the costs for applying different Advanced Configuration and Power Interface (ACPI) states, discuss effects on power saving, and provide an overview on the scope of power saving mechanisms for different x86 processors. To capture software parameters, several changes to existing performance monitoring tools are proposed and implemented. I discuss the general infrastructure and the common components of monitoring and tuning tools in Chapter 5. In Chapter 6, I describe the implementation of several interfaces that enable energy efficiency measurement and tuning. This includes access to power saving mechanisms, the linking of power and performance measurement, and the capability of tools to act on software events. I further describe implementations for each of the interfaces and discuss how they can be used to measure and tune the energy efficiency of HPC applications. I evaluate the tuning implementations on an x86-based HPC system in Chapter 7 and show that the proposed solution can indeed be used to tune parallel applications that use MPI, OpenMP, C++, and Fortran. Chapter 8 concludes this thesis, summarizes the findings, and gives an outlook on future developments in hard- and software.

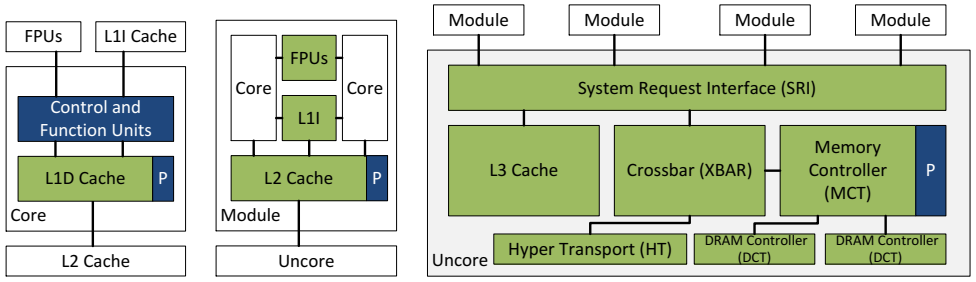
2 Background and Related Work

"Sleep is good", he said, "and books are better.", Tyrion Lannister
A Clash of Kings by George Raymond Richard Martin

Performance and energy measurement and optimization has been a target of scientific work for decades. In this chapter, I first introduce basic terminology and parallel programming concepts in Section 2.1 and Section 2.2, respectively. Afterwards, I examine the aspects of measuring and optimizing performance and energy efficiency. To do so, I split this field into the measurement of performance and power (in Section 2.3 and 2.4) and the optimization of performance and energy consumption (Section 2.5 and Section 2.7). Additionally, I introduce basic concepts of available power saving mechanisms and their implementation in current HPC hardware in Section 2.6.

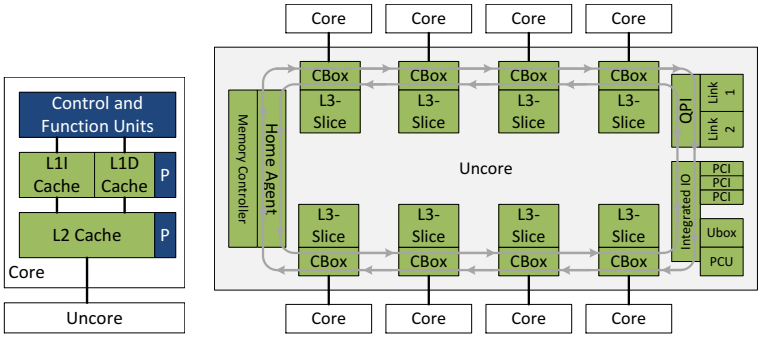
2.1 Hardware Terminology

Terms like processor or CPU have different meanings in the existing literature. To avoid misunderstandings, I define the hardware terminology of this thesis in this section. In my nomenclature, I define a *processor* as a non-separable physical entity that is attached to a mainboard. A processor can host multiple *dies*, where each die is made of a single piece of semiconductor material. Depending on the location of the memory controllers and processor settings, processors form non-uniform memory access (NUMA) [MHSM09] domains, which I call *NUMA nodes*. In addition to the processor inter-connect, processor dies can contain memory controllers, cores, and other devices. I call the union of inter-connect, memory controller, and other devices *uncore*, in accordance with the Intel nomenclature. The AMD term for the uncore is "northbridge", as it represents devices that have previously been external to the processor. Other devices that reside in the uncore include, for example, Peripheral Component Interconnect (PCI) controllers [Int15d, Adv13], uncore hardware prefetchers [Adv13], and caches shared by all attached processor cores. Processor *uncore* are independent processing units that read and execute instructions. They hold a number of computational units, internal data storage, caches, and additional devices like hardware prefetchers that are attributed to specific cache levels. These resources are shared by multiple *hardware threads* if the processor supports hardware multithreading [HP11]. If it does not, a core supports only one hardware thread. In addition, cores can share resources that are not part of the uncore. If a set of cores does share such resources as well as the connection to the uncore, I call the union of cores and core-shared resources a *module* in accordance with the Advanced Micro Devices (AMD) nomenclature [Adv13]. In Figure 2.1 and Figure 2.2, I illustrate the structure of cores, modules, and uncores of AMD Family 15h and Intel Sandy Bridge processors, respectively.
I call the set of all devices that share the same physical address space and run the same operating system (OS) instance a *compute node*. Multiple compute nodes can be connected via network interfaces to communicate with each other and to be able to solve algorithms in a distributed way. However, in such a case, memory has to be transferred explicitly via input/output (I/O) interfaces. I call a constellation of connected compute nodes a *computing system*. A schematic illustration of compute node and computing system is depicted in Figure 2.3.



(a) Core design (b) Module with cores and shared resources. (c) Uncore (northbridge) design. The individual components are connected using an extended star network topology, where the hubs are SRI and XBAR.

Figure 2.1: AMD Family 15h processor architecture [Adv13]. Blue blocks labeled "P" mark prefetchers.



(a) Core design. (b) Uncore design. The Level 3 cache is shared among all cores. The different components are connected by a ring network.

Figure 2.2: Architecture of Intel Sandy Bridge-EP processors according to [Int12a, MHS14]. Blue blocks labeled "P" mark prefetchers.

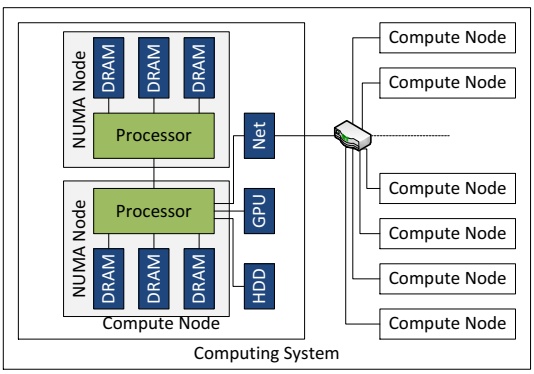


Figure 2.3: Schematic illustration of computing system, compute node, NUMA node and processor

2.2 Parallel Programming

In this dissertation, I focus on the monitoring and optimization of HPC systems and workloads. Thus, I make use of specific parallelization paradigms and techniques, like the process parallel MPI and the thread parallel OpenMP paradigm. Before I introduce these, I describe different types of parallelization and provide an overview on parallel programming paradigms. To do so, I use the terms *thread* (or *software thread*) and *process* as defined by the Portable Operating System Interface (POSIX) standard in [Pos, Section 3.404] and [Pos, Section 3.210], respectively.
According to Flynn's taxonomy [Fly72], computer architectures can be characterized as

- single-instruction stream, single-data stream (SISD),
- single-instruction stream, multiple-data stream (SIMD),
- multiple-instruction stream, single-data stream (MISD), or
- multiple-instruction stream, multiple-data stream (MIMD).

SISD represents a single core, single processor, single compute node computing system, which can process only one date at a time. SIMD architectures comprise vector-processors, array processor, and SIMD ISA-extensions like Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX). Here, a single instruction of a thread can process multiple data items. MISD is an *"uncommon architecture [...] considered to be empty"* according to Antonov [Ant14, Section 1.2.1]. Dongarra and van der Steen state in [DvdS12, Chapter 2] that *"no practical machine in this class has been constructed, nor are such systems easy to conceive"*. Im MIMD architectures, multiple instructions can be processed simultaneously. This can, for example, be implemented with multi-core processors or multi-processor computing systems. All current computing systems listed in the Top500, but also each of their single compute nodes, represent MIMD architectures. Thus, I will limit this work to parallelization concepts for this class. A special case of MIMD programming is single-program, multiple-data (SPMD) [AB09, Section 24.2.2], which was first mentioned in 1984 by Frederica Darema, according to [Dar01]. Here, the same program is executed on all participating hardware threads.
Depending on the target architecture, parallelization concepts can be distinguished into thread-parallel, process-parallel, and hybrid-parallel (both, thread and process-parallel). These are described in more detail in the following paragraphs. Another class of parallelization paradigms is the offloading of work to accelerator devices, e.g., field programmable gate arrays (FPGAs) or GPGPUs. These accelerator devices can only be used in a master-slave context, where regular processor cores issue work to them. More information on this topic can be found in the thesis of Juckeland [Juc12]. In my thesis, I focus on the optimization of workloads for general purpose processors. If however, the accelerator devices implement the same power saving techniques, the proposed concepts of this thesis can be extended to such devices.
Thread parallelism requires a shared address space as defined in [Pos, Section 3.210]. It therefore targets parallelism within a compute node. Thus, the number of parallel executed instruction streams is limited to the available hardware threads within a node and the number of parallel threads that a single operating system instance supports. One basic parallelization library under the GNU Linux platform are Pthreads (POSIX threads). The pthread library enables programmers to create and close threads and to synchronize between them. A new thread gets a data pointer and a function pointer as input variables. Thus, it can be used to implement task-parallel or data-parallel programs. It has been defined under the standard IEEE 1003.1c-1995. As such, the pthread library builds the base for other thread parallel implementations like various OpenMP implementations [Fre16, Int16g], the Boost library [Sch11, Chapter X], or the ISO C++ thread abstraction [ISO14, Section 30.3].
For scientific computing, OpenMP [DE98] is a widely used standard for thread parallel programming. OpenMP is based on a set of compiler directives and a runtime library that implements the calls that are introduced to the software by the compiler. The threads that execute the parallel regions can be scheduled

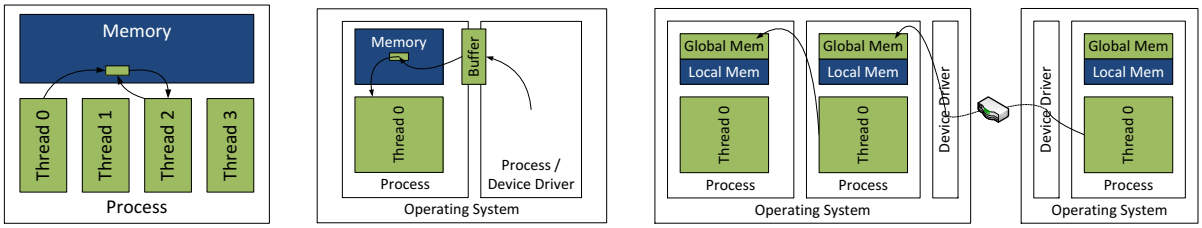
to process loops in parallel, to process tasks, or to process work according to their individual ID. Most compilers [Fre16, Int16g, Por16, Pat16, Ora14] support OpenMP to a certain extent. Since version 4.0, OpenMP supports offloading of computations to accelerators [LYdS+13]. Still, thread parallelism is restricted to a single address space [Pos, Section 3.5], which limits the amount of memory that is available to the threads. In addition, the computing performance is restricted by the number of processor cores within one compute node. Process parallelism does not suffer these limitations and is therefore required for large scale computing.

Process-parallel programming paradigms include message-passing and partitioned global address space (PGAS) strategies. While both can also be applied in thread-parallel scenarios, their concept can also be used by processes since they do not rely on a single address space.

In message-passing based parallelization, processes use messages to communicate data between each other. Hence, data is not shared between the processes, but data copies are sent from one to another. The most common standard used in HPC is the Message Passing Interface (MPI) [mpi15]. MPI defines a set of functions to implement SPMD programs. To distinguish the different participating processes within the code, every one of them is assigned a unique rank. The individual processes can share data by communicating with other ranks explicitly or by performing global operations within a group of ranks. From its first definition in 1994 to the now most current version 3.1, MPI has been revised several times to add, for example, concepts for one-sided communication, dynamic process management, and parallel I/O. MPI *"has now more or less become the de facto standard"* for distributed memory parallel computing, according to Dongarra and van der Steen [DvdS12] and the *"... dominant approach for the distributed-memory, or message-passing, model ..."*, according to Lusk and Chan [LC08]. As I have shown in Chapter 1, typical process-parallel benchmarks that are used to rate HPC systems use MPI as the programming paradigm.

Another parallel programming model that can be applied to distributed memory systems is partitioned global address space (PGAS). According to De Wael et al. [WMF+15], Culler et al. [CDG+93] have been the first to mention this concept. In PGAS, each participating process holds a part of a globally shared memory space. Processes are able to access this shared memory space of all other participating processes via one-sided communication, i.e., without the active participation of the process whose memory is accessed. This is depicted in Figure 2.4c. De Wael et al. give an overview on existing PGAS languages and the concepts in [WMF+15]. PGAS implementations, however, are not part of this dissertation. Still, the concepts that I introduce in Chapter 3 can also be used for such a one-sided access.

Differences between thread, message-passing, and PGAS parallelization are depicted in Figure 2.4. Process-parallel and thread-parallel concepts can be combined [LC08] to make use of the advantages of both. The most common approach is the mixture of the two de-facto standards MPI and OpenMP.



(a) Thread parallel synchronization via shared memory. The individual threads of a process share a memory segment that can be used for communication and synchronization.

(b) In process parallel paradigms like MPI, processes communicate with each other by establishing buffers that receive data from other processes. The buffer can be accessed from a process within the compute node or from a remote process via the network device driver.

(c) In PGAS languages, the memory of a process is separated into a shared (global) and a local segment. Other processes can access the shared segment. To access the memory of remote processes in distributed memory systems, the infrastructure has to support Remote Direct Memory Access (RDMA).

Figure 2.4: Schematic illustration of different thread and process parallel programming paradigms

2.3 Performance Measurement

In [Jai91, Chapter 3], Jain describes three different approaches to performance analysis: analytical modeling, simulation, and measurement. Each method has specific advantages and shortcomings. While analytical modeling is described to be the fastest and most cost-effective technique, it is required to be done by analysts and can thus not be automatized. Simulations can, like analytical models, also be applied before a physical system is available. At a higher cost, they also provide a higher accuracy and scalability, according to Jain. However, the process of simulation needs an existing simulator for the simulated hardware and software. Unfortunately, hardware vendors do not disclose all properties of their products. Thus, component behavior has to be measured and published by scientists. I authored and co-authored a number of publications that target such low-level performance information required for models, e.g., for cache and memory bandwidths and latencies [MHSM09, MSHM11, MHS14, MHSN15] as well as the influence of power-saving features and software on performance and power consumption [MHSM10, HSM+10, SHM11, SHM12, SMW14, SKM13, HSI+15]. Some of the latter work will be presented in follow-up chapters and build a basis for the proposed energy efficiency optimizations.
The third performance analysis method, measurement, has two major short-comings, according to Jain: First, it can only be applied when the hardware (or a prototype) is available. The second drawback are the higher costs compared to the other methods. These are dominated by the necessary hardware acquisition costs. In this thesis, I focus on the measurement and tuning of the software/hardware environment of existing HPC systems. Thus, the hardware is already available, which voids the first shortcoming, and the acquisition costs have already arisen, which reduces the applicability of the second drawback.
Jain lists *"instrumentation"* as *the* tool for performance measurement in [Jai91, Table 3.1]. He further distinguishes this into the measurement techniques *"tracing"* and *"sampling"* [Jai91, Section 8.1.1], where the former is done using *"explicit hooks"* or the (*"trace mode of the processor"*). In contrast, sampling uses *"the system timer facilities ... [to] record ... at periodic intervals"*. The term tracing, however, can have multiple meanings and also describe a type of data aggregation. Thus, I use the term "instrumentation" as defined by Ilsche et al. in [ISSH15]. Ilsche et al. base their work on the PhD thesis of Juckeland, who defines performance analysis as a three staged process in [Juc12, Chapter 3], where sampling and instrumentation (which he calls *"event triggers"*) are techniques for the first stage: *data acquisition*. The three stage concept, which Ilsche et al. take up in their work, defines the following two stages as *data recording* and *data presentation*. Ilsche's nomenclature, which I use in this document, is depicted in Figure 2.5.
In the data recording stage, the data that is produced when the data acquisition interrupts the program is stored. The available techniques comprise summarization and logging. In logging, all gathered information that is available at a point of data acquisition is stored. In summarization, multiple events are

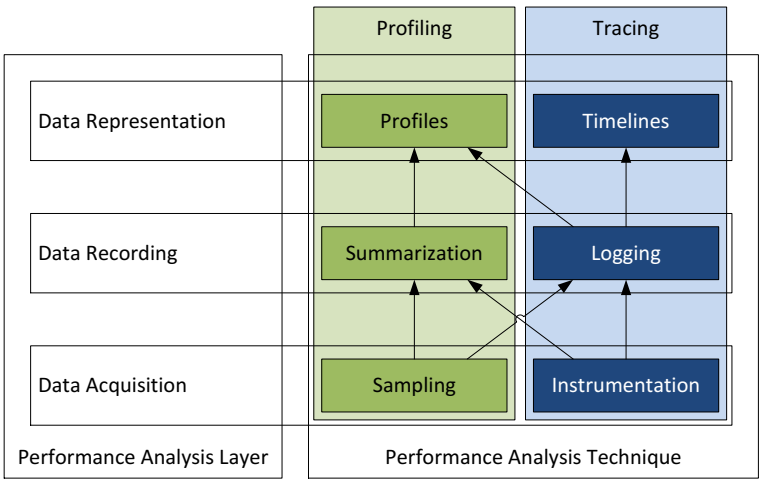


Figure 2.5: Classification of performance analysis techniques by Ilsche et al. [ISSH15, Figure 1]

Table 2.1: Performance monitoring tools

Performance tool	Data-Acquisition	Data-Representation
Score-P [KRaM+12]	Instrumentation, Sampling	Profiles, Timelines
VampirTrace [MKJ+07]	Instrumentation, Sampling* [ISSH15]	Profiles, Timelines
Scalasca 1.x [GWW+10]	Instrumentation	Profiles
Extrae [Bar15]	Instrumentation	Timelines
HPCToolkit [ABF+10]	Sampling	Profiles, Timelines
Open|Speedshop [SGM+08]	Instrumentation, Sampling	Profiles, Timelines
TAU [SM06]	Instrumentation, Sampling	Profiles, Timelines
Alinea MAP [JBOO15]	Sampling	Profiles
CrayPat [KH03]	Instrumentation, Sampling	Profiles, Timelines
VTune [Rei05]	Sampling	Profiles, Timelines
strace [Hal06, Section 13.4.1]	Instrumentation	Timelines
ltrace [Bra07]	Instrumentation	Timelines
perf [Wea13]	Instrumentation, Sampling	Profiles, Timelines
oprofile [Lev03, Lev04]	Sampling	Profiles

|* Prototypes

merged while the performance of a program is measured. This merging procedure results in statistical information about the real events that triggered the recording. This reduces the number of stored event information significantly. The final stage, data presentation, targets the representation of the collected data to the user or a supplementary parsing tool. This comprises profiles and timelines. Profiles can be created with logging and summarization, and present the collected information without a temporal relationship between the events. Timelines, which can only result from logging, do provide this information.

In the following paragraphs, I will present some of the numerous performance measurement tools targeted at parallel applications. While some of them are deemed to be scaling to thousands or even hundreds of thousands of cores, others make better use of the interfaces provided by the most common operating system on HPC machines. I provide an overview on the tools that are described in the next paragraphs in Table 2.1.

Score-P [KRaM+12] is a measurement infrastructure developed by Technische Universität Dresden, Forschungszentrum Jülich, Technische Universität München, and the University of Oregon. It provides instrumentation support for most parallelization methods like MPI, OpenMP, and CUDA. Score-P can be used to create profiles and traces and successes VampirTrace and Scalasca. There are a number of different performance analysis tools that are able to display profiles and traces gathered with Score-P, most prominently Vampir [BWNH01] and Cube [SKVM15]. VampirTrace [MKJ+07] is a predecessor of Score-P. It is focused on instrumentation of parallel programming methods and tracing, even though profiles were supported. Ilsche et al. show the general possibility to extend it with support for sampling [ISSH15]. Scalasca 1.x [GWW+10] is another predecessor of Score-P targeted at profiling instead of tracing. Newer versions of Scalasca only provide performance analysis tools but use Score-P for measurement. Extrae [Bar15] is a measurement infrastructure developed at the Barcelona Supercomputing Center. It provides instrumentation for most parallelization paradigms and targets tracing rather than profiling. Extrae traces can be viewed with Paraver [PLCG95]. HPCToolkit [ABF+10] is a sampling based tool targeted at HPC codes. It supports different parallel programming methods and provides tools to analyze the resulting output. Open|Speedshop [SGM+08] is another open source performance measurement toolsuite that is developed by the Krell Institute. It supports sampling and instrumentation. Additional data acquisition methods can be implemented as Collector Plugins, which are loaded at runtime and inserted into the monitored application via dyninst. Gathered data is written to a communication daemon, which processes it further. The Tuning and Analysis Utilities (TAU) [SM06] support profiling and tracing of parallel programs with instrumentation and sampling methods. Allinea MAP [JBOO15] is a proprietary tool by Allinea. It uses sampling to generate profiles from MPI and OpenMP parallel C and C++ programs. The samples stored in the profiles can be related to source code lines, so a user can see

which lines are executed how often. CrayPat [KH03] is a performance analysis tool created by Cray. It provides support for some parallel programming methods (i.e., MPI, OpenMP, Coarray Fortran (CAF), Pthreads and SHMEM) and supports sampling as well as instrumentation. However, it is a proprietary tool only available on Cray HPC systems. VTune [Rei05] is an Intel proprietary performance analysis tool that allows the user to sample parallel programs and see performance issues alongside the code. More information on HPC targeted performance measurement tools can be found in [ISSH15].
Instrumentation tools for Linux include strace [Hal06, Section 13.4.1] and ltrace [Bra07], which allow the tracing of ptrace [KMPP07] events or calls to dynamically loaded libraries, respectively. Instrumentation via kernel probes [Kri05, MPK06] and user probes [KMPP07] can be used with perf [Wea13]. perf has been introduced with a dedicated system call with Linux 2.6.31 and can also be used to sample a program or a system. oprofile [Lev03, Lev04] also supports sampling but depends on an additional kernel module that has to be installed. More information on Linux based performance measurement tools can be found in [SSIH14].

2.4 Power and Energy Modeling and Measurement

In addition to performance measurement, power consumption information is needed to determine the energy consumption of a monitored workload. In a paper I co-authored, Ilsche et al. define five quality criteria for power measurement infrastructures [IHG+15, Section 1]: *"temporal and spatial resolution, accuracy, scalability, cost, and convenience"* We also claim that these criteria partially contradict each other. A measurement infrastructure with a better accuracy, for example, mostly increases the costs. In this section, I describe the most prominent power measurement infrastructures used in HPC and classify them, including their weaknesses and strengths. This section is based on five scientific papers that I authored or co-authored. In [HIS+13] by Hackenberg et al., we investigated the accuracy of different power measurement infrastructures. In [STD+14], my co-authors and I discussed several aspects of energy efficiency measurement and tuning. In [HIS+14] by Hackenberg et al., we present High Definition Energy Efficiency Monitoring (HDEEM), a measurement infrastructure that is now incorporated in Bull B720 nodes. In [IHG+15] by Ilsche et al., we presented a measurement infrastructure targeted at fine-grained temporal and spatial resolution. In [HSI+15] by Hackenberg et al., we reappraise the Intel Running Average Power Limit (RAPL) energy counter on the Haswell-EP platform.

2.4.1 Modeling Infrastructures and Approaches

Power consumption information can be retrieved from modeling and measurement infrastructures. The idea of modeling power consumption is pursued by various researchers who implement models based on Performance Monitoring Counters (PMCs). These are used to estimate the energy costs of executing different hardware events. The energy costs are later summed up and can then be averaged over time to get an average power consumption. Thus, every monitoring event has to be assigned with a weight that represents the costs for such an event. The costs are determined at an initialization phase where the effect of the measured events on the power consumption is measured for a training-set of workloads. Joseph and Martonosi describe a power model for Intel Pentium Pro processors in [JM01]. In [CM05], Contreras and Martonosi describe such a model for Intel XScale processors. Isci and Martonosi describe a model for Pentium 4 processors in [IM03]. Singh et al. develop a performance-counter based model for AMD Phenom processors in [SBM09]. Goel et al. provide information on how to set-up models for four different processor types from Intel and AMD [GMG+10]. The spatial and temporal granularity of such models is very high, as each processor core provides its own performance counters and these counters are updated every processor cycle. However, the measurement is done internally on the system under test, which influences the result when the sampling rate is too high. The hardware costs are low as no measurement hardware has to be provided. The scalability is high – each new processor can be instrumented and the measurement is distributed over all processors. The convenience can be high when the model is provided in form of a library. The accuracy, however, is low. One reason for this is the

limited number of available PMC registers. For example, if different operations and accesses to different memory locations have different energy budgets, as described by Molka et al. in [MHSM10], each of these has to be counted individually to achieve a high accuracy, which is typically not possible due to the low number of PMC registers. Additionally, the different processor architectures implement only a certain set of events, which further limits the applicability of this approach.

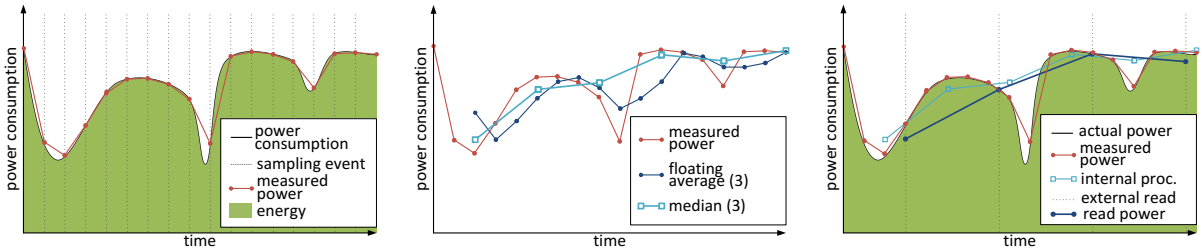
The idea of modeling has also been implemented in hardware by processor vendors. This includes early implementations of Intel's RAPL mechanism [HIS+13], AMD's Application Power Management (APM) [JSK+10], and the power proxy implementation on IBM POWER7 processors [FAWR+11]. Here, a number of processor events is monitored on chip. As there is no limitation due to a restricted number of PMCs, the measurement can be more accurate than software solutions. However, they are not necessarily correct, as Hackenberg et al. describe for RAPL and APM in [HIS+13]. IBM admits, that in the *"... initial attempts to calibrate against hardware, 91 percent of samples fell within plus or minus 10 percent relative error, and 73 percent of samples fell within plus or minus 5 percent error."* Modeled power consumption implemented in x86 processors can be read at a 1 ms temporal granularity for Intel RAPL and with 10 ms on AMD's APM. This reduces the applicability for measuring small code regions. Haehnel et al. describe in [HDVH12] how this limitation can be evaded by adding a known workload for the remainder of the power measurement cycle. However, their approach can significantly increase the runtime of a program and ignores the affect of other influences like temperature on the power consumption for the fill-in workload. Finally, the Intel implementation RAPL lists the consumed energy of a certain region of a processor, which is, like PMCs, monotonically increasing. To handle overflows correctly, i.e., to detect each of them, Intel recommends reading this counter at least once every minute. AMD reports the average power over the last measurement period with APM. Thus, information can be retrieved with one reading, instead of the two needed for RAPL to determine the difference. However, in APM, every measurement has to be collected to compute the power consumption over longer time-periods.

2.4.2 Measurement Infrastructures

If the focus of the power instrumentation is on accuracy, actual measurements are the better choice. Here, flaws in the model can be ruled out. However, as Jain argues in [Jai91, Section 3.1], a measurement is also not necessarily correct. In this section, I describe common practices in power measurement instrumentations and their advantages and shortcomings.

Most of the accurate power measurement infrastructures provide two sampling frequencies, an internal, and an external one. The internal one is used to actually measure the power consumption, the external one defines at which rate the data is being made available for external reading. In between, the collected data can be filtered to avoid incorrect measurements. Thus, the accuracy is limited by the filter algorithm, the used instrumentation method (including the measurement hardware), the internal and external sampling frequency of the instrumentation infrastructure, and the frequency of the external tool reading this information. The whole process is depicted in Figure 2.6.

The measurement infrastructure uses an instrumentation technology like measurement shunts, capacitors, or hall-sensors to determine the current power consumption of a device [IHG+15]. In some instrumentations, only the current is measured and the voltage is expected to be constant over time, which lowers the accuracy of the instrumentation additionally. The next limitation stems from the discretization of the power consumption, based on the internal sampling rate. If this sampling rate is too low to cover fluctuations in the power consumption, information can be irreversibly lost. Afterwards, most systems filter the sampled data to discard wrong measurements (measurement faults) and reduce the amount of collected data. The resulting data can later be read by external tools. However, these tools are most often not synchronized with the internal process of providing new data to be read. Thus, the timing information of the collected data is less precise. Additionally, the filtered samples are sampled another time, which further reduces the resolution and the accuracy.



(a) Internal measurement – loss of information: time discretization

(b) Internal processing – loss of information: filtering, possible consolidation of measurements

(c) External processing – loss of information: lower update rate, read rate not synchronized with provision rate

Figure 2.6: Loss of information in power measurement infrastructures over three phases: internal measurement, internal processing, external measurement. Based on [HIS+14, Fig. 1]

General Power Measurement Infrastructures

Several researchers implement low-level instrumentation to analyze the power consumption of single components. For example, Ilsche et al. implement an infrastructure with a high temporal and spatial resolution in [IHG+15]. They instrument two different systems with hall effect sensors and shunts to direct current (DC) lines and components. They sample the power consumption with National Instruments Data Acquisition (DAQ) cards at a sampling rate of up to 500 kSa/s. Ge et al. describe PowerPack in [GFS+10]. They add resistors to several DC pins and use a National Instruments (NI) input module to collect the data. Penguin Computing and Sandia Labs present PowerInsight in [LPD13]. They use riser cards and Molex adapters with small Hall effect sensors and sample with a sampling rate of 1 kSa/s. Bedard et al. present PowerMon2 in [BLFP10]. The infrastructure is able to instrument up to eight DC channels with a measurement resistor and sample with up to 1,024 Sa/s. Likewise, ARDUPOWER can also be used to instrument DC channels with up to 5,880 Sa/s, as Dolz et al. describe in [DHK+15].
At a lower spatial and temporal granularity, researchers often use calibrated power meters like Watts up? Pro [Wat] or ZES ZIMMER LMG [ZES11]. These can easily be connected to the AC input of the computing system and provide data with a sampling rate of up to 20 Sa/s. The internal sampling rate, however, is significantly higher.

Power Measurement Infrastructures by Vendors

Power consumption is also of interest for data center monitoring. It can be used to implement power caps and to find hot-spots. Thus, vendors implement different infrastructures for power measurements. One conventional implementation is the usage of an instantaneous power measurement within the Power Supply Unit (PSU). The data is later accessed via Intelligent Platform Management Interface (IPMI) from the Board Management Controller (BMC). Vendors like Bull [HIS+14], Cray [FCGS14], and IBM [KFH+14] implement solutions with higher spatial and temporal resolution. With the "Haswell" microprocessor generation, Intel also shifts the RAPL mechanism from modeling to measurement, according to Hackenberg et al. [HSI+15].

APIs for Power Measurement Data

In [KHN12], Kluge et al. present dataheap, which can be used to collect the data from multiple inputs. Clients can access the data post-mortem or live. There are multiple data sources that can provide dataheap with power information. These include sources for NI-DAQs, ZES ZIMMER power meters, processor internal power models, PSUs, and the Bull power measurement infrastructure HDEEM [HIS+13, HIS+14]. Alternatively, PMLib also supports a client server model where multiple data sources like commercial PDUs, professional power meters, and low-level instrumentation as stated by Llopis et al. [LDGB+15]. The Performance API (PAPI) enables a direct access to solutions from processor and GPGPU vendors [WJK+12]. The PowerAPI [BNRS13] targets to unify the access to different power sources. However, it does not only cover the measurement of power but also describes interfaces to change system parameters that influence the energy efficiency, like the processor frequencies.

2.5 Tuning Parameters and their Classification

In this section, I describe different tuning stages for hardware and software tuning. In the following, I use the word *tuning* for approaches that improve the performance (e.g., the throughput or the energy efficiency) in comparison to the default configuration. The term *optimization* is used, if the goal of the tuning is to reach a local or global performance optimum.

2.5.1 Tuning as an Optimization Problem

Even though a variety of objectives can be the target of a tuning or optimization process, the general idea is the same for all of them. Such optimization problems and how they can be solved are, for example, described by Rothlauf. He defines six phases for solving the optimization process in [Rot11, Chapter 2.1 Solution Process]:

> *“ (1) recognizing the problem, (2) defining the problem, (3) constructing a model of the problem, (4) solving the model, (5) validating the solutions, and (6) implementing one solution ”*

These can be mapped to the performance evaluation phases that Jain describes in [Jai91, Section 2.2]. In this section, I focus on the third and fourth phase, the creation and solving of a model. When creating a model, one has to define variables that have an influence on the outcome of the optimization. These $(x_1, ..., x_n)$ define a vector of decision variables, where each of these can have a number of values. According to Rothlauf, the available settings for these variables can be continuous or discrete, which influences how the optimization model should be solved. Furthermore, he defines that an assignment of specific values to the decision variables is stated as x, which he calls a solution, and the set of all possible solutions as X. To optimize the model, the solutions and an evaluation function are needed. This function assigns a specific real value to a single solution. This value is to be optimized by finding the best settings for each x_i. To limit the complexity of long vectors of decision variables, Rothlauf also states a widely accepted model, which takes similarities between different solutions into account [Rot11, Equation 2.3]. The evaluation function and a search algorithm can be used to find an optimum. However, most search algorithms either end in local optimums or have a high time-complexity, based on the range and the number of decision variables. Still, several tuning approaches in HPC use such search algorithms to find optimums and tune applications, e.g., AutoTune [MCS+13], and ATLAS [WD98], which are described later in more detail.
Another problem is that multiple optimization targets can contradict each other, for example, if the two targets are a low energy consumption and a low runtime. To limit the number of solutions in such cases, the concept of pareto optimality [Luc08] can be helpful as the range of the decision variables is reduced. I distinguish between software and hardware tuning methods, which are described in Section 2.5.2 and Section 2.5.3, respectively. I also consider the time when the tuning can be applied as well as the scope.

2.5.2 Software Tuning

I distinguish software tuning methods into: (1) Source Code Tuning, (2) Compiler Tuning, (3) Linker Tuning, (4) Binary Tuning, (5) Parameter Tuning, and (6) Runtime Tuning. I depict examples and the software stage when each of these is applied in Figure 2.7.
Source code tuning includes manual changes in the algorithm as well as optimizing source-to-source compilers like Scout [KFMPN12] that automatically improve the algorithm's implementation. A control-loop that validates the performance result of the compiled source code is used in ATLAS [WD98] and the FFTW library [FJ05] to select the best variant of an algorithm for a given hardware. Compiler tuning is done by applying optimizing compiler flags or choosing a different compiler. Tools that use loops, which measure the performance of a resulting binary for different compiler flags, include the compiler flag tuner from AutoTune [MCS+13] or OpenTuner [AKV+14]. While linking the object file resulting from the compiling process, another optimization can be used by linking a different version of a library

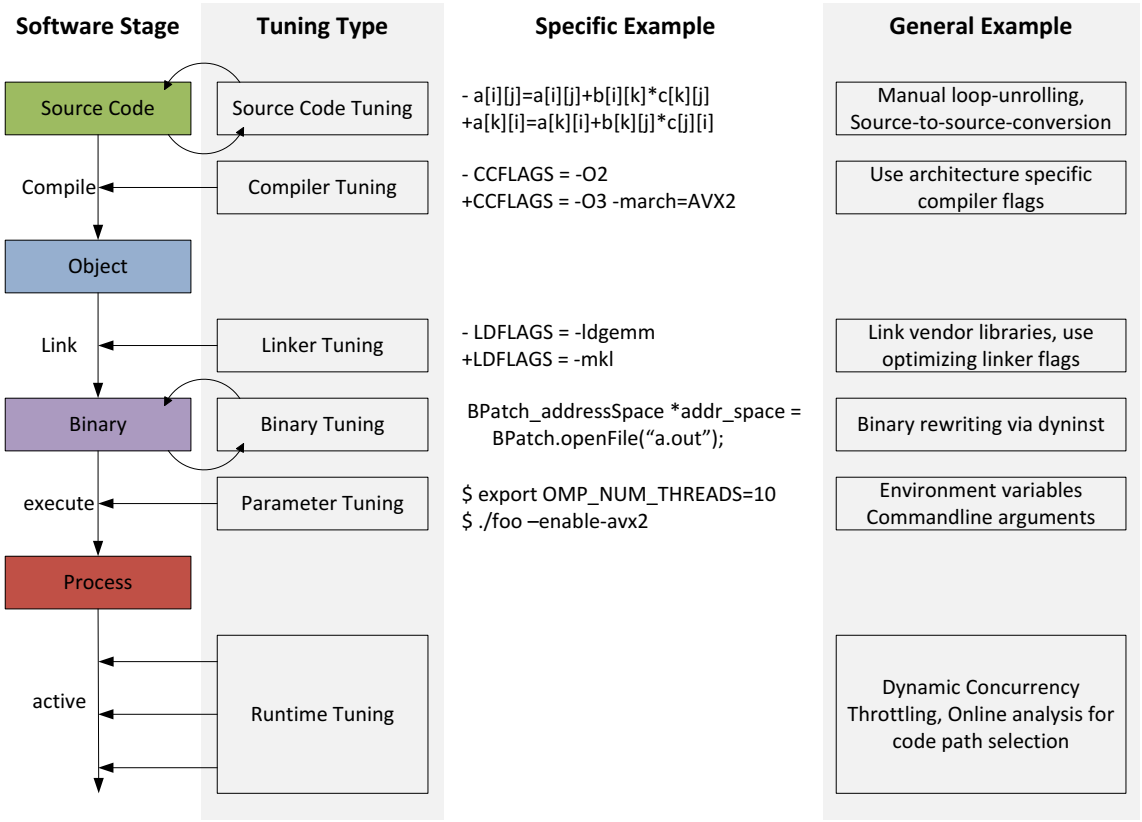


Figure 2.7: Types of software tuning, including examples

or applying linker parameters that enable optimizations. The resulting binary can then be modified with binary tuning. With software like Dyninst [BH00], PIN [LCM+05], and Dynamo [BDB00], applications can be patched to change their behavior, e.g., when a function is entered and exited. A more common technique is the usage of environment variables or command-line arguments to enable users to fine-tune programs. These are highly dependend on the runtime environment or the binary that is to be optimized. In the following, I describe some of these parameters that can be relevant in the HPC context.

There are different implementations of the MPI standard from hardware vendors [Int15e] as well as Open-Source implementations [GLDS96, GFB+04]. The available tuning parameters depend on the MPI version and the underlying hardware that is used. E.g, buffering or splitting of small or large messages is handled transparently to the user or programmer. The exact limits for such optimizations can usually be fine-tuned. One example is the "eager limit", which is available for different libraries. There are several tools that support the tuning of such parameters [Int16f, PFJM10, PPF12].

Some of the tuning parameters are defined by the OpenMP standard, e.g., the number of parallel threads to use [Ope15, Section 4.2] and the scheduling of loop iterations [Ope15, Section 4.1]. Other optimizations depend on the implementation, e.g., the behavior of idling threads, which is called spin count or block time. Such a parameter exists in the libgomp (GNU), Intel, Open64, and PathScale OpenMP runtime environment. Depending on the runtime, the environment parameter is called `GOMP_SPINCOUNT`, `KMP_BLOCKTIME`, `O64_OMP_SPIN_COUNT`, or `PSC_OMP_THREAD_SPIN`. Depending on the spin-count setting, the threads used by the OpenMP runtime actively wait (poll) for new parallel regions. A lower spin-count enables power-efficiency optimization, as inactive threads do not need processor resources. A higher spin-count typically reduces the latency to assign work to an existing thread.

The Intel Math Kernel Library (MKL) [Int16e] uses runtime parameters to define the behavior of thread-parallelism. For example, the environment variable `MKL_DOMAIN_NUM_THREADS` can be used to define the number of threads for the domains Basic Linear Algebra Subprograms (BLAS), Fast Fourier Transform (FFT), Vector Mathematical Functions (VML), and Parallel Direct Sparse Solver Interface

(PARADISO) and the internal memory management can be controlled via `MKL_FAST_MEMORY_LIMIT` to limit the usage of Multi-Channel DRAM (MCDRAM) on Xeon Phi processors.
The GNU C-library glibc [LDS+93] provides environment variables to fine-tune its memory allocation scheme. The environment variables `M_ARENA_MAX` and `M_ARENA_TEST` can be used to set the maximal number of memory arenas (memory pools that can serve threads in parallel).
The operating system can be considered as another software with multiple tuning parameters. GNU/Linux provides a huge set of parameter and runtime tuning options. Parameter tuning[1] options are applied when a kernel module is loaded or when the kernel boots. When the operating system is running, it still provides the possibility to tune parameters via `sysfs` entries, system-calls, or `sysctl` interfaces. Additionally, the operating system assigns resources, for example memory, to single applications. The location of the individual memory pages on NUMA nodes also has a significant influence on the application's performance [MSHM11]. Thus, Linux provides interfaces to efficiently assign virtual memory to specific NUMA nodes [Kle05] or migrate pages later [Lam13]. Another tuning factor that relates to hardware resources is scheduling. To avoid an inefficient usage of the available processor cores, Linux implements a system call for pinning software threads to a set of hardware threads [Lov13, Chapter 6].
The tuning stage and the validation stage do not necessarily coincide, but the tuning has to be applied at an earlier stage. For example, a compiler tuning that targets throughput still needs to be linked and executed to validate whether the tuning has been effective. A compiler tuning that targets a lower binary size can be validated after compiling.

2.5.3 Hardware Tuning

Hardware tuning can, like software tuning, be applied at different stages. I distinguish the types (1) Architecture Definition, (2) Layout Tuning, (3) Process Tuning, (4) BIOS Tuning, and (5) Runtime Tuning. The single stages are described in Figure 2.8.
In the first stage, the hardware is designed. Here, three different tuning types are applied that interact with each other. This includes the *architecture definition*, where the incorporated elements are defined, e.g., the type and size of caches of a processor or the number of PCI slots on a mainboard. In *layout tuning*, the elements defined in the *architecture definition* are arranged. This includes, for example, VHDL optimization routines for processors. On a coarser hardware scale, the same is true for compute nodes, which includes decisions about the mainboard layout or the cooling infrastructure. The third tuning

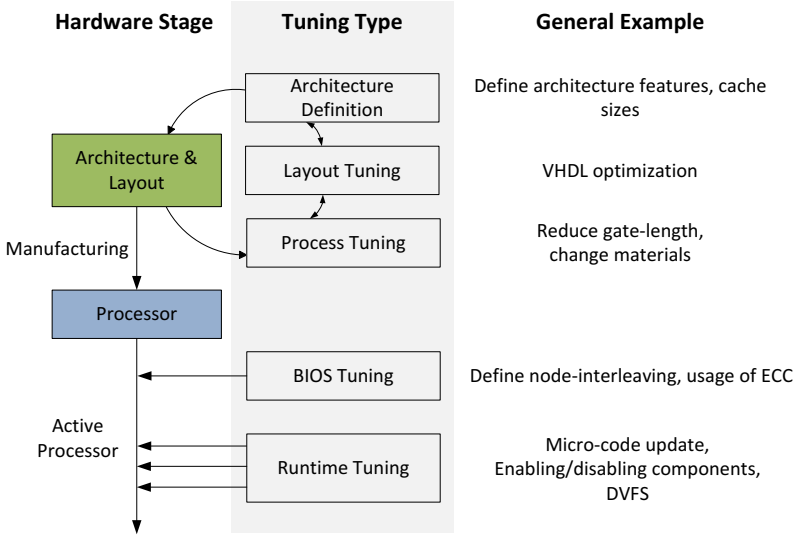


Figure 2.8: Tuning types of hardware, including examples

[1]The whole list of kernel parameters is available online at https://www.kernel.org/doc/Documentation/kernel-parameters.txt

type represents changes in the manufacturing process and is called *process tuning*. This refers to all optimizations that are applied to the material used to build the processor or other hardware components. This includes the reduction of gate-length as well as the doping of semiconductors with other materials on processor scale and advances in voltage regulators, cooling elements, and other devices on computing system scale. The result of process tuning has implications on the architecture definition and the layout. When more transistors are available due to a process shrink, for example, additional resources, like larger caches, can be incorporated, which changes the processor layout. Therefore, these three tuning steps cannot be dissociated from each other.

However, like software parameters, the architecture and layout process targets a broad range of customers and is not optimized for each single one of them. Thus, processor manufacturers implement fine-tuning mechanisms that can be applied by the BIOS or at runtime. These parameters are typically defined in the respective processor manuals [Int15a, Adv13, Int15c]. Their configuration, however, is often realized via Model Specific Registers (MSRs) that can only be manipulated with privileged access, i.e., in an operating system context. Operating systems can handle such tuning parameters internally (e.g., by using power saving mechanisms if available [PLB07]) or provide interfaces that can be accessed by privileged users, e.g., via `sysfs` entries [PS06]. In Section 6.1, I describe an interface to access such parameters with fine-grained rights management.

2.5.4 Additional Tuning Distinctions

Tuning methods can also be distinguished into *online* and *offline* tuning, with respect to the scope of the analysis phase. For online tuning, a running group of hardware or software elements is analyzed. Thus, every online tuning can be considered a runtime tuning. However, if the rules for runtime tuning are created after a process has finished, this runtime tuning is to be seen as offline runtime tuning. An additional criterion of distinction is the scope of the analyzed hardware or software that is to be optimized. For example, parallel programs can be monitored at thread-level, process-level, node-level (all threads and processes within a compute node) or over several compute nodes. I call the minimal scope (one software or hardware thread) a *local* tuning and the maximal scope (all used software or hardware threads in the computing system) a *global* tuning.

2.6 Power Saving Mechanisms of Computing Systems

In the previous section, I classified different tuning parameter types. This section introduces specific hardware mechanisms that can be used to lower the power consumption of computing devices.

A computing system consists of several components that draw power. However, not all of them can be the target of power optimizations due to a lacking support for such actions or due to missing interfaces to trigger these. In the following chapter, I describe what influences the power consumption of two main power consuming components: processors and DRAM, and which power saving techniques these components support.

To describe the effects of power saving mechanisms in state-of-the-art hardware, I introduce models that describe the power consumption thereof based on several input parameters starting with processors. These models can be distinguished according to their different level of abstraction. Electrical engineers focus on the integrated circuit that uses power to switch transistors. Such a low-level approach can help to determine general relationships between electrical input parameters and the overall power consumption. Weste and Harris describe the most established model in [Wes11, Section 5.1.3]:

$$P_{total} = \overbrace{\overbrace{\alpha C V_{DD}^2 f}^{P_{switching}}}^{P_{dynamic}} + P_{short\text{-}circuit} + \overbrace{(I_{leak} + I_{cont}) \times V_{DD}}^{P_{static}} \tag{2.1}$$

In Equation 2.1, P_{total} is the total power consumption of the circuit, $P_{switching}$ describes the power consumption for switching transistors, $P_{short\text{-}circuit}$ is the short-circuit power consumption. *"In nanometer processes [...] short-circuit current has become almost negligible"* according to Weste [Wes11, Section 5.2.5]. P_{static} is determined by fabrication circumstances that have an impact on leakage and contention. The most influential factor for these is the supply voltage V_{DD}. Influential parameters for switching power are α, which describes the switching probability of the processor calculable from the switches of single functions [Wes11, Section 5.2.1.2], the capacitance C, and the clock frequency of the circuit f.
State-of-the-art processors are complex and can comprise several billion transistors where single transistors can be mapped to specific functional units. For these functional units one can describe the circumstances under which the probability of a transistor switch increases. For example, transistors that are used for the L2 cache will not be used/switched when the computation uses data that resides in the L1 cache and transistors within vector units will not be used for scalar integer arithmetic. Thus, computer scientists prefer a high-level approach for such functional units, where they correlate a PMC event to a specific switching probability α. Examples for such models are described by Joseph et al. [JM01], Contreras et al. [CM05], Singh et al. [SBM09] and Goel et al. [GMG+10].
Effects like process variation [DM02, BKN+03], aging, and degradation [AM05, AKVM07] as well as the influence of ambiance parameters like temperature increase the complexity of such a model. Additionally, these effects induce that the model for a certain processor cannot be used for all instances of that processor at all times.
The target of power saving mechanisms at runtime are the dynamic power consumption and, to a lesser extent, the leakage power consumption. The most common techniques are clock modulation, Dynamic Voltage and Frequency Scaling (DVFS), clock gating, and power gating.
David et al. present a model for estimating DRAM power consumption in [DGH+10]. According to them, the most influential factors are the number of bank activations, where a memory row is copied to the row buffer, and the number of reads and writes. In addition, DRAM draws a static amount of power when its clock is enabled.

2.6.1 Power Saving Mechanisms of Processors

In this section, I describe the most common power saving techniques of processors and their influence on processor power consumption. Furthermore, I show how the single hardware techniques are mapped to ACPI states. I detail the implementation of these mechanisms in x86 processors and summarize their usage in other processor families and close with a description of Dynamic Random Access Memory (DRAM) power saving mechanisms.

Dynamic Voltage and Frequency Scaling

Dynamic Voltage and Frequency Scaling (DVFS), which is also known as Dynamic Voltage Scaling (DVS), is a technique that enables changing the processor's frequency and voltage at runtime [CSB92, HGS98, GKL99]. While a decreased frequency f reduces the dynamic power consumption linearly, the reduced voltage V_{DD} lowers the switching power consumption superlinearly and the static power consumption linearly as described by Equation 2.1.
To implement DVFS, vendors have to provide voltage regulators and dynamic clock sources as well as an interface to communicate set frequencies and voltages to these components. Thus, the implementation is more complex compared to clock modulation or clock gating. Examples for DVFS are the implementations of ACPI P-state on recent Intel [Int15a, Section 14.1], AMD [Adv13, Section 2.5.2], Advanced RISC Machine (ARM) [Wat09, Section 8:10], and International Business Machines (IBM) [WRF+10] processors. An illustration of DVFS is depicted in Figure 2.9. If the voltage cannot be changed, Dynamic Frequency Scaling (DFS) can be used alternatively. Here, only the frequency is changed while the voltage remains constant. However, with this mechanism, only minor power savings are possible, since only a part of the dynamic power consumption is reduced.

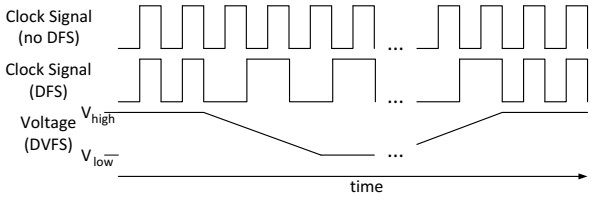


Figure 2.9: Illustration of processor signals and voltage changes for DVFS mechanisms

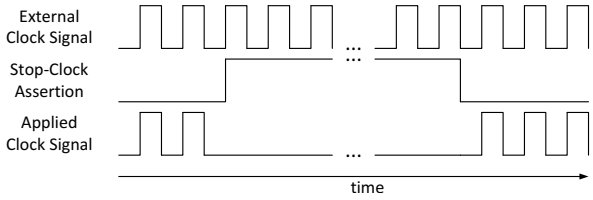


Figure 2.10: Illustration of processor signals for clock gating

Turbo Frequencies

The progress in power saving mechanisms and the integration of many components on a single processor leads to a situation where most of the time, the power budget of a processor is not fully used. This power budget is also called Thermal Design Power (TDP) and defines the thermal energy that the cooling infrastructure must be capable to dissipate. In a worst case scenario, the parameter α is maximal so that the processor's power dissipation $P_{total} = TDP$ when reference frequency and the associated voltage is applied. However, in typical computing scenarios, stalls in the processor cores lower the probability α. Additionally, the number of instructions issued in parallel depends on the workload and seldom matches the number of possible instruction issues. Furthermore, if devices like processor cores are disabled via clock or power gating, which is described in more detail in the following sections, the static and dynamic power consumption of the overall processor is reduced. However, cooling devices are designed for the worst case. Thus, most of the time, the cooling infrastructure is overdesigned for a modern processor. Therefore, contemporary processors may increase their frequency above the reference frequency when certain conditions are met. While this increases the static (due to an increased voltage) and dynamic power consumption (voltage and frequency) of a subset of cores, the performance can also improve.
In newer processors from AMD [Adv13, Section 2.5.2.1.1] and Intel [Int15a, Section 14.9], these turbo frequencies are accompanied by on-die energy modeling or measurement mechanisms that ensure that the power consumption of the whole processor stays within the TDP or even exceeds it for a limited time period. Special registers extend the power monitoring infrastructures to enable system programs to change the targeted power consumption of a processor at runtime.

Clock Gating

Clock gating [QWPW97] is a technique where an integrated circuit or parts thereof can be completely isolated from the clock signal. This is achieved by logically ANDing the processor clock signal with a NEGated stop-clock assertion that indicates a disabled clock gating. This is depicted in Figure 2.10. When activating the stop-clock signal, the processor clock signal f is effectively removed from the circuit. Hence, the dynamic power consumption is reduced to 0. However, when the clock signal is removed, an external signal has to re-enable the circuit. One example of clock-gating is the implementation of the ACPI C1-state on recent Intel processors (e.g., [Int12b, Section 4.1.2]).

Clock Modulation

Clock modulation can be understood as an advanced clock gating mechanism. When a certain condition (clock modulation assertion) is set, the clock is disabled whenever a clock modulation signal indicates it. The resulting signal that is applied to the processor is thus the result of ANDing the external clock signal, and the NEGated result of ANDing the condition, and the clock modulation signal. This is depicted in Figure 2.11. The assertion signal can be enabled by software or hardware mechanisms [Int15a]. An example for clock modulation is the implementation of thermal throttling on Intel processors [GBCH01].

Power Gating

Power gating [MDM+95] is a technique where the input voltage V_{DD} of an integrated circuit is effectively removed. This reduces the power consumption of the power-gated components to zero. One implementation of power-gating is the implementation of deep C-states (e.g., C6) in Intel processors [Int12b].

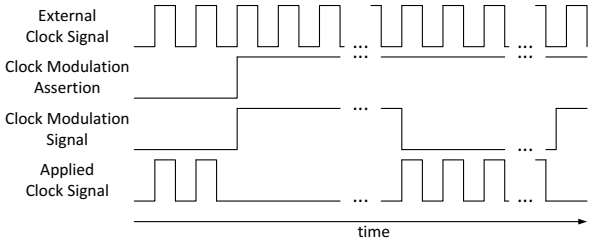


Figure 2.11: Illustration of processor signals for clock modulation

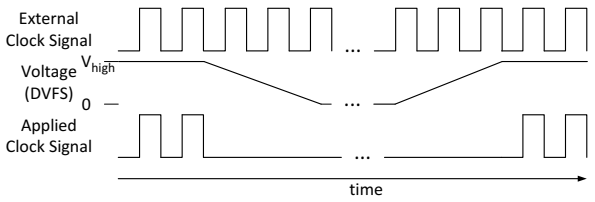


Figure 2.12: Illustration of processor signals for power gating

However, on-chip memory loses its state when the voltage is removed, which has serious implications. Processors or processor cores that use power gating have to save their state, which includes all components that are attached, e.g., caches. When re-enabling the cores, these states have to be restored or reset.

Other Techniques

Some features of modern processors like prefetchers or speculative execution increase the number of transistor switches α to increase performance. However, the increased number of transitions also raises the power consumption. Depending on the quality of the underlying heuristic of a processor feature and the actual pattern of the executed instructions, these can have different influences on energy consumption. Programmers can influence α by reducing data lengths (e.g., by using single precision floating point data instead of double precision if possible).

2.6.2 Standardization of Power Saving Mechanisms

Access to power saving mechanisms is standardized via the ACPI [acp16]. ACPI succeeds the Advanced Power Management (APM) [apm96] API, which defines interfaces between an operating system and a Basic Input/Output System (BIOS) to handle power management. ACPI has been co-developed by Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. Other companies like AMD also implement devices according to the ACPI standards. In 2013, the ACPI standard has been included in the Unified Extensible Firmware Interface (UEFI) portfolio and is now developed by the ACPI Specification Working Group at UEFI. In the following, I refer to specific power management states with their respective ACPI names according to Specification 6.1. Therefore, I shortly introduce these states in this section. The mapping of ACPI states to the previously described power saving mechanism is given in Table 2.3.

According to [acp16, Section 2.6], processors and devices can define up to 255 *performance states* (*P-states*) that can be used when in an active power state. The description of the ACPI objects [acp16, Chapter 8.4.6.2] indicates that processor P-states should be implemented with frequency scaling. The ACPI object also defines a power dissipation and a frequency transition latency that can be used by the operating system to determine an appropriate performance state.

Table 2.3: Power optimization techniques and their impact on power consumption and performance

Technique	Power Saving Effect		Performance Influence	Typical ACPI States
	$P_{switching}$	P_{static}	on scaled component	(see Section 2.6.3)
Frequency Scaling	$\times f_{reduced}/f_{ref}$	none	$\times f_{reduced}/f_{ref}$	P-state, C1E-state
Voltage Scaling	$\times(V_{reduced}/V_{ref})^2$	$\times V_{reduced}/V_{ref}$	none	P-state, C1E-state, C3-state
Clock Gating	reduced to 0	none	reduced to 0	C1 state, C3-state
Power Gating	reduced to 0	reduced to 0	reduced to 0	C6+ state
Clock Modulation	$\times f_{reduced}/f_{ref}$	none	$\times f_{reduced}/f_{ref}$	T-state
Prefetcher Disable	varying	none	varying	-

The ACPI standard defines different *processor power states* or *C-states* [acp16, Section 8.1]. **C0** corresponds to the state of a full working processor - i.e., a processor that executes instructions. The **C1** state shall be supported by a native instruction, maintain caches and need no hardware support from the chipset. Additionally, it is defined to have a low latency, which should not be considered when deciding whether to use this state. The optional **C2** state has a certain latency that is communicated to the OS. The processor has to maintain caches and needs support from the chipset. This could, for example, include support from the voltage regulators to use DVFS in addition to clock gating. The **C3** state has a higher latency than C2. Residing in **C3** allows the processor to ignore memory snoops from the processor bus. According to the standard, the operating system is responsible for maintaining cache coherence. Additional C-states (C4 and higher) have a higher latency than C3.
Additional throttling states (*T-states*) can be used to limit the power dissipation of processors. They are defined in [acp16, Section 8.1 and 8.4.5]. They can be applied in addition to performance states and limit the performance to a specific percentage of the P-state, where the performance is lower for higher T-states.
Devices can also define *device power states* (*D-states*) that have to provide information about the available functionality and the wake-up characteristics, which can then be read and interpreted by the operating system to use D-states efficiently. In the lowest D-state, devices can also implement performance states that are similar to processor performance states. D-states are defined in [acp16, Section A.2]
Systems can implement *sleeping states* (*S-states*), which use different *global system states* (*G-states*) as defined in [acp16, Section 2.2 and 16.1]. In the global system state **G0**, the S-state is always **S0**, the system is active and all the previously mentioned states can be used. To switch to a deeper S-state, the system can change to **G1**, where the state of the operating system is preserved. Depending on the depth of the S-state, the system is allowed to stop processors, disable processors, put DRAM in self-refresh, or disable DRAM. The **S5**-state represents a "Soft-Off" and is related to **G2**. In this state, the power supply unit of the system is still powered, but the system itself is shut-down. In **G3**, the power to the system is removed.

2.6.3 Supported Power Saving Mechanisms on Current x86 Processors

Modern x86 processors provide a broad variety of hardware adaptations [Int15a, Adv13, Int15b]. These include the support for ACPI C-states, P-states, and T-states. In this section, I describe the hardware interfaces for different processors and devices to access these features. In addition to the standardized ACPI state support, contemporary x86 processors provide access to performance critical features that can be toggled or fine tuned. These can be changed via register accesses, usually MSRs for processor core related settings and PCI registers, which are also called Configuration Space Registers (CSRs) for uncore components. While most adaptations are described in processor manuals, others are described in vendor tools (e.g., the Intel BIOS Implementation Test Suite[2]).
To distinguish between core and package C-states, I use the abbreviations *CC* an *PC*, respectively. The term "package C-state" is used by Intel in the respective processor manuals, e.g., [Int13b, Section 4.2.5] and describes an overall processor state where in addition to core C-states, uncore components are set in low-power states. To enter a specific package C-state, all cores of a processor must enter a specific core C-state. For example, to enable PC3, all cores have to reside in CC3 (or higher).

Overview for Intel64 Processors

Intel64 is the Intel implementation of the 64-bit x86 instruction set. The most recent server and HPC implementation code names are Nehalem, Westmere, Sandy Bridge, Ivy Bridge, Haswell, and Broadwell (ordered chronologically). Even though these processors differ in their microarchitecture, the supported ISA as well as the available energy efficiency mechanisms are very similar. They also implement common interfaces to access these mechanisms.

[2] `http://biosbits.org/`

Processor Power States

Intel processors based on the Nehalem, Westmere, Sandy Bridge, Ivy Bridge, Haswell, Broadwell, and Skylake microarchitecture implement at least four core C-states, including *CC0*, *CC1*, *CC3*, and *CC6* [Int11c, Int11d, Int12b, Int14c, Int14a, Int16b]. Some models also implement deeper states like *CC7*.

When a processor core enters the *CC1* state, the cache entries and the architectural state are preserved. The processor core also continues to process snoop requests in order to maintain cache coherence. If all cores in a processor are in the *CC1* state, the complementary *C1E* state is enabled. In that case, voltage and frequency for the whole processor is reduced to enhance the power savings. When a processor core enters the *CC3* state, cache lines are flushed from L1 and L2 caches to the shared last level cache. The core keeps its architectural state and is clock gated. To keep the architectural state, a significantly lower voltage is needed [Wes11, Section 10.4.3]. When the *CC6* state is entered, the architectural state is flushed as well. It is stored in a dedicated Static Random Access Memory (SRAM) in the uncore. Afterwards, the core is power gated. The *CC7* state *"exhibits the same behavior as the* CC6 *state."*[Int14a], but is used to determine the depth of the package C-state.

The processor enters the *PC3* state if all of its cores are in *CC3*. The L3 cache is still *"snoopable"* [Int11c] or *"active"* [Int14a] in the *PC3* state, but memory is put into self-refresh. When all cores are in *CC6*, the package enters the *PC6* state. In that state the L3 cache retains its content but the data is not accessible. Presumably clock gating is used in the control logic and/or data paths in that case. Processors based on Sandy Bridge, Ivy Bridge, Haswell, and Broadwell microarchitectures implement a special *PC2* state to answer requests from devices and other processors that cannot be processed in deeper package C-states. If a processor receives snoop requests or accesses to its portion of DRAM occur while being in *PC3* or *PC6* the processor transitions to *PC2* state for processing the requests. For these processors, the uncore is not only clock-gated in PC3 and PC6, but retention voltage is applied to it [Int12b, Int14c, Int15b, 4.1.2], which has to be ramped up to the voltage of the minimal supported uncore frequency to answer the request. Some desktop processors implement an even deeper package C-state – *PC7*. In this state, some power is removed from portions of the system agent as well [Int14a, Int16c, Section 4.2.5]. Additionally, the last level cache is flushed and power gated when all cores agree to enter *CC7*.

Contemporary Intel processors implement hardware features that allow the processor to dynamically decide which C-state the processor's cores should use. Auto-Demotion is used for switching to a lower C-state than requested by the operating system. Auto-Promotion allows the processor to use higher C-states. This effectively overrides the decisions made by the operating system. In [Int15a], Intel documents MSRs for the configuration of these features. The Linux kernel driver `intel_idle` uses these MSRs to alter the settings previously specified by the BIOS.

Processor Core Performance States

All listed processors implement P-states by using DVFS. The operating systems can handle these via writes to the architectural register MSR_IA32_PERF_CTL (0x199). In addition to the Performance States that clearly define a frequency, contemporary Intel processors additionally support a Turbo mode. Enabling this mode allows the processor to use frequencies higher than the reference frequency if the thermal headroom allows it. The Turbo frequencies also depend on the number of active cores and are declared to the operating system in an additional MSR. In another register, BIOS and operating system can set the minimal frequency that has to be applied to MSR_IA32_PERF_CTL before a Turbo frequency is applied. On most Intel processors, all cores share a single voltage and frequency domain. Only Haswell and Broadwell server processors provide fine-grained performance states per processor core [HSI+15].

Other Performance States

Nehalem-EP, Westmere-EP, Haswell-EP, and Broadwell-EP processors provide an additional voltage and frequency domain for the uncore [HBB+10, GSS15]. The uncore frequency of Haswell-EP processors depends on the available thermal budget and several other aspects. This mechanism is described by Hackenberg et al. in [HSI+15].

Other ACPI States and Adaptable Processor Mechanisms

In addition to processor power and performance states, Intel processors support throttling states, which includes a hardware control loop that applies T-states in times of temperature induced stress [Int15a, Chapter 14.5]. The processors also support suspend states and memory self-refresh, which is used in the S3 state.

Beginning with Sandy Bridge processors, Intel implement a power capping mechanism called *Running Average Power Limitation* RAPL [Int15a, Chapter 14.7] on their processors. With this mechanism, BIOS and operating system can set up a specific power limit and time window. The processor is then forced to stay within the power limit – averaged over the time window. As the power consumption of the processor components depend on various factors, the frequency over time is not predictable. However, the internal power measurements that are used for the capping mechanism are also exported via incrementing registers.

Additionally, Intel processors implement components like prefetchers and the Energy Performance Bias (EPB) that influence performance and energy efficiency. Prefetchers in Intel processors monitor memory access patterns in the L1 and L2 cache and request data from deeper memory levels beforehand. The mechanisms are detailed in the architecture specific sections of [Int16a]. The EPB enables BIOS, operating system, and administrators to shift the optimization target of the processor between a lower power consumption and a higher performance. Gough et al. provide a section about the EPB implementation on Haswell server processors in [GSS15, Chapter 8]. All of these options can be changed and triggered via writes to MSRs in order to influence the power consumption of processors.

Overview for AMD Family 15h Processors

AMD processors of the 15h family support different hardware adjustments that can be changed at runtime. These include various frequency settings [Adv13, Section 2.5.2.1], idle states [Adv13, Section 2.5.3], prefetcher settings [Section 3.5, Section 3.14][Adv13], and other hardware settings. With the implementation of new features in the “Piledriver” (model 0x10-0x1F) processor generation, more options have been made accessible [Adv15]. However, while Intel processors clearly define how processors implement the available ACPI states, AMD allows to tune these via MSRs and uncore registers in the BIOS and runtime tuning stage (see Section 2.5.3). Examples about what can be tuned are given in Table 2.4 and Table 2.5.

Processor Power States

AMD family 15h processors support up to three core C-states [Adv13]. Each of them can be configured by writing to PCI registers during the initialization of the processors. The C-state specifications are encoded in three 16 Bit registers as described in Table 2.4. In *CC1* (recommended settings: 0x000B), the compute unit does not apply a frequency divisor, i.e. the frequency is defined by the P-state prior to entering the C-state. Consequently, the frequency does not need to be ramped up to answer incoming probe requests. After approximately 407 μs, the L1 and L2 caches are flushed. Furthermore, the cache

Table 2.4: AMD family 15h model 00h-0Fh C-state control register [Adv13]

Bit	setting	options
0	direct probe	frequency used to handle probes 0b: use frequency defined by P-state 1b: adhere to clock divisor (see 7:5)
1	cache flush	0b: disabled, 1b: enabled
3:2	flush timer	select timer register 01b: D18F3 0x0DC, 10b: D18F4 0x128
4	reserved	n/a
7:5	clock divisor	000b: disabled, 001b: 2, 010b: 4, 011b: 8, 100b: 16, 101b: 128, 110b: 512, 111b: turn off clock
8	power gating	0b: disabled, 1b: enabled
15:9	reserved	n/a

flush success monitor can be used to record the rate of cache flush timer expirations relative to C-state exits. If the success rate is higher than the configured threshold (D18F4 0x128[20:18]), caches are flushed before the timer expires. However, the feature is disabled (D18F4 0x128[20:18] = 000b) by default. Power gating is not used in *CC1*. The *CC6* state (recommended settings: 0107h) uses the same frequency settings. However, it also applies power gating. Therefore, the operating frequency is only relevant until caches are flushed and the compute unit is powered down completely. *CC6* uses a separate timer register for the cache flush, but the specified timeout equals the *CC1* setting. The cache flush success monitor is not supported by the selected timer register. The processor states of all cores that go to *CC6* are stored in the DRAM.

AMD family 15h processors with a model number $\geq$ 0x10 introduce northbridge power states that make use of memory self-refresh, and northbridge clock and power gating [Adv15, Section 2.5.4.2]. These require all cores to use C-states. Flags for the different northbridge power state features are added to the C-state control registers. Thus, they can be configured independently for every C-state.

Processor Performance States

Like processor power States, performance states are highly configurable on AMD family 15h processors. To tune these, processors implement a definition register for each of the eight different P-states (see Table 2.5). In these registers, BIOS and operating system can change the frequency and voltage definitions for each performance state. However, two processor cores within a computing module share one frequency setting. Even though both can specify their own P-state, the minimal P-state for both cores is used by the hardware for the module. While each computing module has its own frequency, all of them share the same voltage [Adv13, Section 2.5.2.3.4], based on the minimal P-state set for any module.

Other Performance States

In addition to the P-states targeted at voltage and frequency reduction, AMD family 15h processors also implement Boost States [Adv13, Section 2.5.2.1.1]. These enable a higher frequency than P-state 0 when the available thermal budget is not exhausted. The number of Boost States and regular P-states is limited, since each of these states has its own definition in a specific MSR (see Table 2.5). To determine the current power consumption and the difference to the TDP, AMD implements a power model based on processor internal events [JSK+10, HIS+13]. The uncore of the processor implements up to two P-states [Adv13, Section 2.5.2.2], high and low. The active *"northbridge P-State"* depends on the Core P-states, which means that it is defined by the lowest active core P-state. However, the mapping of core and northbridge P-states can be changed at runtime by writing to MSR registers (see Table 2.5). On AMD family 15h processors with a model number $\geq$ 0x10, northbridge P-states also define an associated memory P-state [Adv15, Section 2.5.7.1]. A memory P-state defines a frequency that is applied to the memory and timings for the DRAM accesses.

Table 2.5: AMD family 15h model 00h-0Fh P-state register [Adv13]

Bit	setting	options
5:0	CpuFid:core frequency ID	Specifies the core frequency multiplier. Frequency = 100 * CpuFid/divisor (see CpuDid)
8:6	CpuDid: core divisor ID	Specifies the core frequency divisor. divisor = 1 « CpuDid
15:9	CpuVid: core VID	Voltage ID for this P-State. Presumably the ID used for communication with the voltage regulators.
21:16	read as zero	n/a
22	Northbridge P-State	1=Low performance NB P-state. 0=High performance NB P-state
... *(more options)* ...		

Other ACPI States and Adaptable Processor Mechanisms

AMD processors also support suspend states (e.g., [Adv13, Section 2.5.5.1]). Additionally, they implement memory self-refresh, which is used in the S3 state. Furthermore, they support a throttling of DRAM activity to limit the power dissipation of DRAM banks. They also provide prefetchers at three different levels, according to [Adv13, Section 3.10, 3.21]. The "Data Cache Configuration" and "Combined Unit Configuration" MSRs describe hardware prefetchers that reside in the L1 cache and the shared L2 cache of each processor core and module, respectively. The DRAM prefetcher resides within the memory controller of the processor, has a table size of 32 and can differentiate between prefetch and core requests. Its configuration is defined in the PCI registers 0x11C and 0x1B0 of the corresponding memory controller PCI devices. The configuration for this prefetcher is highly sophisticated and thoroughly described in [Adv13].

2.6.4 Supported Power Saving Mechanisms on Other Processor Families

Other processor families also support the power saving mechanisms that are presented in this section. IBM POWER processors starting with POWER7 support per-core P-states via DFS and two different C-states: "Nap" and "Sleep" [FAWR+11]. Since POWER7+, IBM processors also implement Turbo frequencies [ZFD+15]. POWER8 processors implement the per-core P-states by using DVFS. ARM described a processor with DVFS and clock gating support in 2005 and added power gating in 2006 [Wat09]. In 2007, ARM described a set-up with three different voltage domains [Wat09, Figure 165].

2.6.5 Power Saving Mechanisms for DRAM

DRAM that implements the DDR2, DDR3, or DDR4 standard supports three different power saving techniques: self-refresh, power-down and Clock Enable (CKE) low [Ass09, Section 3.10, 3.11, 3.12]. The CKE signal can be set to low to stop the DRAM clock. This is comparable to the stop clock signal for processor clock modulation that is used in C1 and T-states as described earlier in this chapter. When CKE low is applied, the clock frequency of the DRAM can be changed. If the DRAM does not process any commands and all memory banks are precharged, the memory can enter power-down mode. However, memory content can be lost, if power is not applied before the refresh interval ends. To avoid this, DRAM modules can support a self-refresh mechanism. An internal timer applies power to the memory module before the refresh interval ends. Afterwards, the memory module can switch back to power-down. If the DRAM does not support self-refresh (or if it is disabled), the memory controller can issue the signal within the refresh interval. However, in such a case, the memory controller cannot be turned off, which lowers the power saving potential.

2.7 Power-based Energy Efficiency Runtime Tuning

The power saving mechanisms described in the previous section are the target of many different tuning attempts. In this section, I present a short overview on energy efficiency tuning strategies and classify them according to the targeted computing system. I distinguish between *performance-based energy efficiency tuning* and *power-based energy efficiency tuning*. The former increases energy efficiency by reducing the runtime, the latter by using hardware power saving mechanisms. In this thesis, I focus on power-based energy efficiency tuning, since performance-based energy efficiency tuning is more or less a by-product of performance tuning, which has been a research topic for a much longer time period.

2.7.1 Energy Efficiency Metrics

In 2007, The Green Grid published two metrics [BRPC07] to define the efficiency of a data center: Power Usage Effectiveness (PUE) and its reciprocal datacenter infrastructure efficiency (DCiE). These can be

used to describe how efficient a data center uses power. The PUE compares the power supplied to the data center to the power that is used by its computing components and thus describes the overhead for infrastructure, e.g., cooling. In 2012, The Green Grid refined the metric [AAFP12] to specify details.

Voltage converter inefficiencies within compute nodes, however, cannot be attributed by the PUE metric as these are part of the computing infrastructure. This problem also affects other node internal devices like processor fans. Thus, Patterson introduced the Total-Power Usage Effectiveness (TUE) metric in 2013 [PPH+13] that targets to describe the power consumption overhead for internal components that do not contribute to computing.

However, in this thesis, I focus on the energy efficiency of software running on an HPC system. Here, the most commonly used metrics are "computations per watt", energy to solution (ETS), and energy-delay product (EDP). Computations per watt describe the computing performance that can be achieved within a given power budget. The most prominent example is the Green500 list [FS09], which reports the energy efficiency for running the Linpack benchmark for HPC systems listed in the Top500 list [Meu08] using GFLOPS/Watt as the metric. This metric is also intuitive to describe systems that are bound to a certain power-limit. The reciprocal metric is called energy per operation, which is used to describe the costs for single processor events. For example, Molka et al. [MHSM10] use it to describe costs for single instructions and memory references. Others use such metrics for power consumption models. David et al. [DGH+10] assign specific weights with this metric to different DRAM operations. To characterize the energy efficiency of whole workloads, the metric ETS can be used. This is done, for example, by various SPEC benchmarks [MBB+12, JBC+15]. Energy, however, does not cover all expenses of the ownership of a computing system. To emphasize the throughput to a larger degree, the energy-delay product (EDP) metric was introduced in 1994 [HIG94]. For scenarios where performance is even more important, the delay part may be emphasized by using an ED^nP metric [BBS+00], e.g. ED^2P , or the generalized FTTSE metric [BC10], where an arbitrary function defines the ratio of performance and energy consumption.

2.7.2 Existing Energy Efficiency Runtime Tuning Infrastructures

In addition to the various criteria described in Section 2.5, power-based energy efficiency tuning can be distinguished according to the used power saving mechanism. In the following, I name some prominent examples of energy efficiency tuning infrastructures targeted at different areas of application. Table 2.6 provides an overview on the classification of the described mechanisms.

Table 2.6: Overview of different power-based energy efficiency runtime tuning tools

Tuning Tool	Data Acquisition Type	Data Acquisition Scope	Analysis Scope	Power Saving Mechanism	Online/ Offline
ondemand gov. [PS06]	Sampling	hardware thread	hardware thread	P-state	online
cpuidle menu gov. [PLB07]	Instrumentation	hardware thread	hardware thread	C-state	online
Green Governors [SKK11]	Sampling	hardware thread	hardware thread	P-state	online
pe-governor [SH11]	Sampling	hardware thread	hardware thread	P-state	online
Adagio [RLdS+09]	Instrumentation	MPI process	MPI process	P-state	online
LaBaTa [Mül13]	Instrumentation	MPI process	global	P-state	online
Renci/UNC [BPP15]	Instrumentation	MPI process	global	T-state	online
Green Queue [TLP+12]	Instrumentation	MPI process	global	P-state	offline
ACTOR [CMBAN08]	Instrumentation	OpenMP thread	process	C-state	online
ENAW [WSM15]	Instrumentation	OpenMP thread	process	P-state	online
Periscope Tuning Framework [MCS+13]	Instrumentation	software thread (central infrastructure)	global	plugins, e.g., P-state	offline
SLURM [Sch]	Instrumentation	computing system	global	S-state	online
Marlowe [GJW11]	Sampling	computing system	global	S-state	online

Tuning at Compute Node Granularity

Linux offers access to power saving mechanisms of processors [PS06, PLB07]. It does not only support the ACPI interfaces and offer access, but also implements tuning tools that target an efficient usage of these features. For example, the *cpufreq* kernel infrastructure supported two frequency governors that optimize the P-state based on different heuristics in its initial implementation. Additionally, whenever a CPU is idle, the *cpuidle* kernel infrastructure automatically selects a C-state according to several heuristics. These standard approaches are far from optimal, which is discussed in the following paragraphs.

Standard frequency governors use a metric that is not applicable for an effective usage for current processor generations. First of all, they are unaware of the usage of C-states. A *race-to-idle*, where processor cores use a high frequency to finish the computing as fast as possible and enable the system to idle for the remaining time, can be more efficient – depending on the power consumption of the processor and the other system components. The next flaw of the governors is that they are unaware of possible architectural bottlenecks. The roofline model [WWP09] describes that the provision of additional resources does not necessarily increase the performance. A performance gain can only be achieved by providing additional resources to the current bottleneck. Contrariwise, resources of non-bottleneck components can be withdrawn as long as the bottleneck is not shifted. Thus, if the memory-controller is the bottleneck, the resources in terms of voltage and frequency for processor cores can be reduced. In [SH11], I described a possible alternative governor that uses performance counters to assess the memory boundedness of codes. I evaluated the theory of a general applicability of such an approach in [SHM12].

The cpuidle infrastructure uses several heuristics to compute a probable idle time. This includes the length of the last idle periods, the DMA latency defined by the system, and possible timers that are set to expire in the next future. Based on that expectation, the idle menu governor uses the deepest C-state that is available and whose reported latency is lower than the expected idle time. In [SMW14], I have shown that the propagated wake-up times that processors provide via ACPI are not necessarily accurate.

Shutting Down Compute Nodes

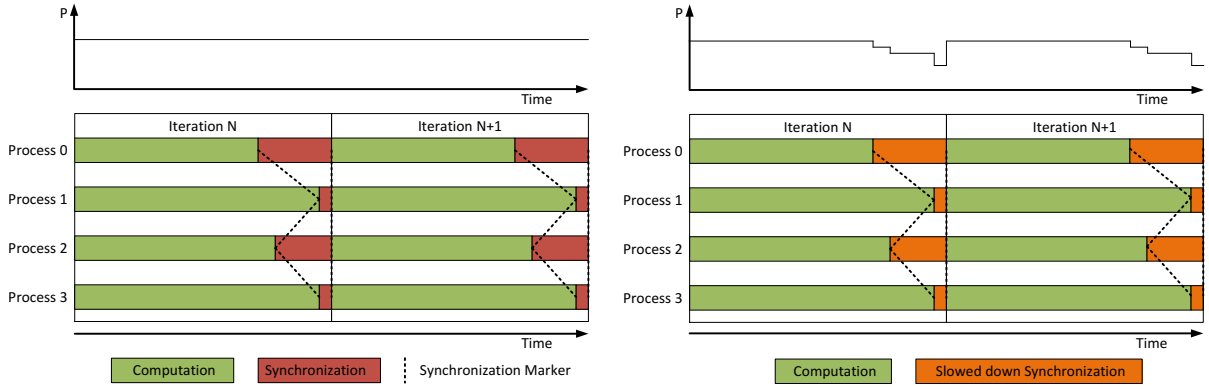
When multiple compute nodes are available to be managed by a single batch system scheduler, this scheduler can power down these nodes in times of low occupation using ACPI S-states. This technique is available for HPC clusters [Sch] as well as in data centers [GJW11]. When the monitored workload intensity increases above a certain threshold, the scheduler can reactivate these nodes, e.g., via Wake-on-Lan [Bui06].

Tuning of HPC Applications

In HPC, the usage of standard parallelization paradigms like OpenMP and MPI and the SPMD principle that runs comparable workloads on a number of hardware threads make it easier to tune applications for energy efficiency. Given tuning software can be differentiated into region-based and balancing-based approaches.

Region-based tuning, as depicted in Figure 2.13, targets specific code regions, for example synchronization routines. When these functions are executed, a tuning mechanism makes use of ACPI states to lower the power consumption without decreasing performance significantly. The scale of the recognized region depends on the implementation of the tuning mechanisms. AutoTune [MCS+13], for example, recognizes one region for its optimization, typically the main function. Other approaches target the synchronization regions where the performance is reduced without hurting the computing performance. For example, Bhalachandra et al. use T-states for optimizing the energy efficiency of synchronizing MPI routines [BPP15].

Balancing-based tuning is based on the idea that paths in a parallel program that are not on the critical path can be slowed down until they become the critical path. Various researchers implement such a balancing algorithm for global barriers. Müller uses P-states to balance MPI applications in [Mül13]. He



(a) Parallel processes synchronizing at the end of each iteration

(b) Optimization targeted at synchronization regions

Figure 2.13: Region-based optimization. In the depicted example, power saving mechanisms are used while synchronization routines are active. This reduces the power consumption of the computing system.

describes and implements a library that can be called when a global synchronization of an imbalanced workload is executed.

Rountree et al. take this approach a step further with Adagio [RLdS+09]. They distinguish separate regions by their call stack and determine the imbalance for each computation phase that is followed by a synchronization phase. They then use P-states to slow down computation phases that are deemed to be off the critical path. This is depicted in Figure 2.14. While Rountree et al. determine the critical path at runtime, Tiwari et al. use an offline approach for their optimizations [TLP+12]. In [WSM15], Wang et al. use P-states to increase the energy efficiency of imbalanced OpenMP parallel programs.

A totally different approach for energy efficiency tuning is the usage of Dynamic Concurrency Thottling (DCT), as described by Curtis-Maury et al. in [CMBAN08]. Here, the number of parallel threads in a thread-parallel program is reduced if a parallel region does not profit from additional threads. With a reduced number of threads, less hardware threads and less cores are needed for the computation and more cores can use C-states to lower the overall power-consumption. In a further work, Curtis-Maury et al. extend their approach to a hybrid use of P-states and C-states on hybrid parallel applications for slowing down critical paths and reducing concurrency [CMSB+08].

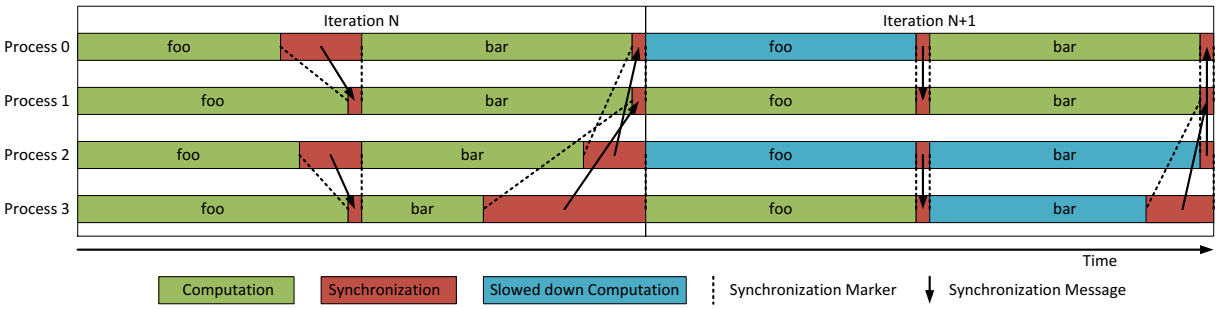


Figure 2.14: Balancing-based optimization targeted at re-balancing imbalanced workloads with support for multiple phases. In the depicted example, a tuning algorithm determines the imbalance of functions foo and bar in the Nth iteration. Based on that information, the performance of processes that are not on the critical path is reduced in the following iteration.

2.8 Conclusion

In this section, I introduced the nomenclature that is used throughout this thesis to classify different hardware components. Furthermore, I listed and classified parallel programming paradigms used in HPC computing. I also described different state-of-the-art performance and power measurement infrastructures. At the beginning of my dissertation, there had been no tool that incorporated both, scalable performance measurement and power logging capabilities, which is a requirement for an energy efficiency evaluation tool. However, the different programming languages and parallelization paradigms used in HPC and the various power measurement infrastructures for different system and processor vendors indicate a high implementation effort to create such a tool from scratch. Therefore, I extended scalable performance measurement infrastructures and incorporated the possibility to add associated metrics to the existing information. This can be used to determine energy-efficient system configurations and test whether a tuning has been successful.
In Chapter 3, I describe how energy-efficient configurations for HPC applications can be found. This relies partially on hardware parameters, which are described in Chapter 4. In addition to that, the application has to be analyzed, which includes a survey of its performance and energy efficiency saving potential. This, however, requires a monitoring tool that is able to capture performance and power information. In Chapter 5, I show how the additional information can be classified and assigned to the application performance logs. Furthermore, I discuss common parts of measurement and tuning infrastructures and describe implications for a unification of both. Based on this, I demonstrate two extensions for existing performance measurement infrastructure in Chapter 6. One of them can be used to incorporate power information, but is generic enough to also cope with other metrics. The second interface extends an available measurement infrastructure to implement an alternative handling of the arising application events. This enables me to implement energy efficiency tuning extensions that can make use of an infrastructure, which is able to cope with most common programming languages and parallelization paradigms. In Chapter 7, I use the infrastructure to determine energy-efficient hardware configurations and tune the energy efficiency of HPC applications. To demonstrate the flexibility of my approach, I present a region-based offline tuning and a balancing-based online tuning.

3 A Model for Power-based Energy Efficiency Tuning

> *"Is this another 'I think we have a potential energy flow' kind of problem?"*, Sophia Coloma
> **The Human Division** by John Scalzi

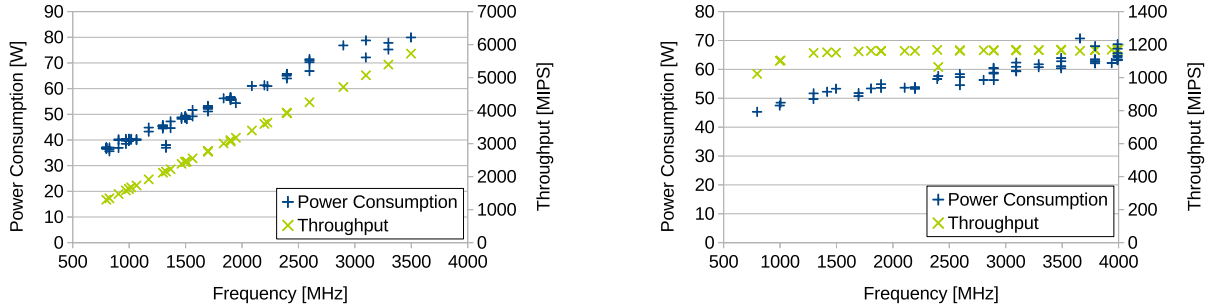
Each power saving mechanism has a certain influence on the power consumption of the tuned component and the runtime of the executed software. This is depicted in Table 2.3 in Section 2.6. However, this table only provides an abstract overview and does not provide any information when a certain mechanism should be used. Therefore, another model has to be used that takes the actual parameters from a real system into account. In this chapter, I describe such a model that can be used to quantify the effect of different power saving mechanisms on energy efficiency. However, the model does not cover parameters like temperature dependent power consumption, which depends on the surrounding environment and previous power dissipation. I published an initial description in [SIB+16].
In this model, a program is executed on a specific hardware. The hardware supports a number of power saving mechanisms. Each combination of these mechanisms is one configuration c of the hardware, the default configuration is c_0. Please note that any configuration can be defined as default. This could, for example, be a predefined reference setting (e.g., the reference processor frequency) or the result of an algorithm that finds a single optimal configuration to the whole application. The program S is a sequence of regions $r \in R(S)$ that have different performance characteristics and power demands. The runtime of a specific region with a specific hardware configuration c is defined as $t(r, c)$, the power consumption of the hardware that executes the region as $P(r, c)$. The runtime depends on the amount of instructions $I(r)$ that are to be executed in the region and the region's throughput $p(r, c)$ in instructions per second. The configuration of a region is either set before the region is entered ($c \leftarrow c_r$), or it is inherited from the previous region ($c \leftarrow c_\prec$). Depending on the weighting function $\lambda(r, c) = f_\lambda(P(r, c), t(r, c))$ (e.g., $ETS(r, c) = P(r, c) * t(r, c)$), each region provides an energy efficiency value $e(\lambda, r, c)$ that defines its efficiency compared to the default $e(\lambda, r, c_0) = 1$ (see Equation 3.1). The relative efficiency of two settings c_α and c_β can be compared by using the times $t(r, c)$ or the throughputs $p(r, c)$ (if f_λ is the product of P^n and t^m ($n, m \in \mathbb{R}$)), as described in Equations 3.2 and 3.3. The throughput substitution can be used for common metrics like energy to solution (ETS) and energy-delay product (EDP) and is valuable when multiple regions with the same behavior are executed.

$$e(\lambda, r, c) = \frac{f_\lambda(P(r, c_0), t(r, c_0))}{f_\lambda(P(r, c), t(r, c))} = \frac{f_\lambda(P(r, c_0), \frac{I(r)}{p(r,c_0)})}{f_\lambda(P(r, c), \frac{I(r)}{p(r,c)})} \tag{3.1}$$

$$\frac{e(\lambda, r, c_\alpha)}{e(\lambda, r, c_\beta)} = \frac{f_\lambda(P(r, c_\beta), t(r, c_\beta))}{f_\lambda(P(r, c_\alpha), t(r, c_\alpha))} \tag{3.2}$$

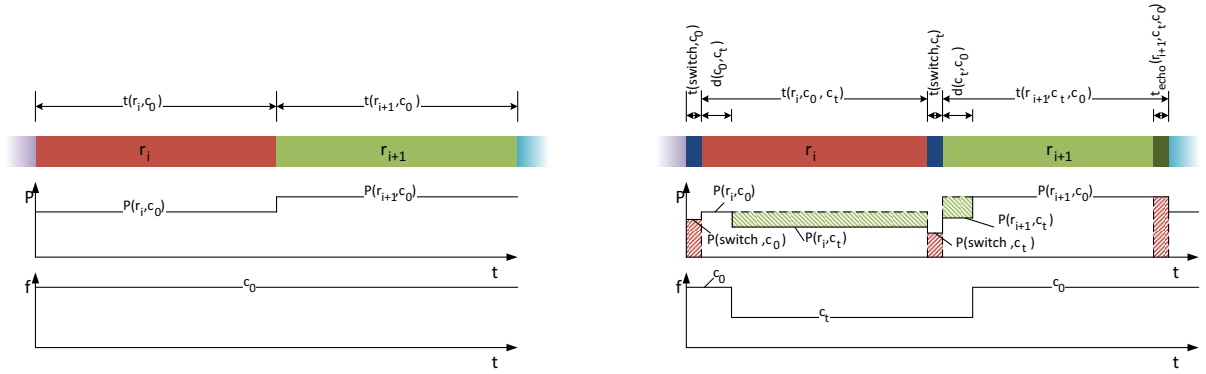
$$\frac{e(\lambda, r, c_\alpha)}{e(\lambda, r, c_\beta)} = \frac{f_\lambda(P(r, c_\beta), \frac{1}{p(r,c_\beta)})}{f_\lambda(P(r, c_\alpha), \frac{1}{p(r,c_\alpha)})} \text{, if } f_\lambda = P^n * t^m | n, m \in \mathbb{R} \tag{3.3}$$

The grouping of regions with a common behavior is discussed in Section 3.1. I call such region groups functions. In Figure 3.1, I depict two different functions of the OpenMP parallel benchmark BT which is part of the NPB suite [BBB+94]. While the efficiency of one region (Figure 3.1a) is optimal at the highest frequency, the other region (Figure 3.1b) can be optimized when λ is ETS or EDP, since the throughput is not affected as long as the frequency is above 1300 MHz.
In the simplest case, depicted in Figure 3.2a, a single configuration c_0 is used over the whole program run. When a specific region r_i is about to be tuned by applying a different configuration c_t that is reset afterwards, runtime, throughput, and power consumption of r_i and its successor r_{i+1} are significantly



(a) `!$omp do @z_solve.f:42` (b) `!$omp parallel @rhs.f:17`

Figure 3.1: Model parameters for two different parallel regions of the NAS Parallel Benchmark BT (OpenMP, Class C, HTT used, 8 Threads, Intel Core i7-6700K, varying frequency).



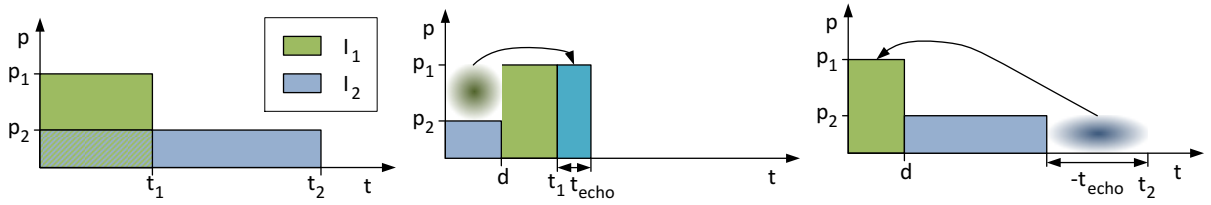
(a) Execution of two regions r_i and r_{i+1} with default configuration c_0. r_i has a lower power demand. The configuration (frequency) is constant.

(b) The configuration is set to c_t before r_i is entered, which decreases the power consumption of r_i. The new configuration is applied with a certain delay d. After r_i ends, the default configuration c_0 is restored. Please note that $t_{echo}(r_i, c_0, c_t)$ is 0 in this example.

Figure 3.2: Comparison between an unoptimized and optimized execution of r_i.

influenced. This is depicted in Figure 3.2b. Before the tuned region is started, the configuration is changed. Here a supplementary interface enables the software to change the configuration. Accessing this interface has its own runtime $t(switch, c)$ and power consumption $P(switch, c)$ that depends on the configuration that is active before the change is applied. In this model, it is assumed that the new configuration is only applied after the software finished the change request. This takes transition latencies of P- and T-states into account. After a specific delay $d(c_\prec, c)$, the hardware actually applies the new configuration. In the following, throughput, runtime and power consumption of r_i may change. In the given example, the configuration is reset to c_0 after r_i is completed. Thus, another access to the configuration interface is necessary. This access has a runtime of $t(switch, c_t)$. Runtime and power consumption of the successive region r_{i+1} are also influenced from the optimized configuration c_t. In Figure 3.2b, the throughput of r_{i+1} is decreased while the hardware is still in c_t. Thus, the runtime increases by a specific amount t_{echo}. This prolongation can be calculated by the throughput and runtime of r_{i+1} under c_0 and c_t and $d(c_t, c_0)$.

Each region has a specific number of instructions $I(r)$ that are executed. Depending on the throughput $p(r, c)$ of the used configuration one can expect a certain runtime $t(r, c)$, since $t(r, c) = \frac{I(r)}{p(r,c)}$. This is depicted in Figure 3.3a. If a part of the runtime of the region is executed with a different configuration (e.g., r_{i+1} in Figure 3.2b is initially executed with configuration c_t instead of c_0), the runtime changes. If the throughput of the other configuration is lower, then less instructions have been performed over a period of time and these have to be executed afterwards (refer to Figure 3.3b). If the throughput is



(a) Two different configurations (c_1, c_2) with different througputs p have varying runtimes t for the same number of instructions I in a specific region

(b) If the initial cofiguration has a lower throughput ($c_\prec = c_2, c = c_1$), the runtime increases with respect to the time of the target configuration

(c) If the initial cofiguration has a higher throughput ($c_\prec = c_1, c = c_2$), the runtime decreases with respect to the time of the target configuration

Figure 3.3: Explanation of the runtime increase in r_{i+1} from Figure 3.2b

higher, the instructions that remain after applying the correct configuration can be executed faster, as is described by Figure 3.3c. Here, t_{echo} is negative. Equation 3.4 describes how $t_{echo}(r, c_\prec, c)$ performs under different throughput levels for the previous configuration $c_\prec$ and the target c.

$$t_{echo}(r, c_\prec, c) = \frac{p(r,c) - p(r,c_\prec)}{p(r,c)} * d(c_\prec, c) \tag{3.4}$$

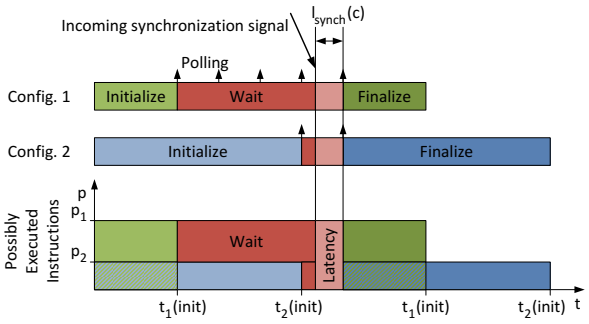
The access to change the configuration increases runtime and energy consumption additionally. The total runtime and energy of a region r, when the configuration is switched from $c_\prec$ to c, is described in Equation 3.5 and Equation 3.6.

$$t(r, c_\prec, c) = \overbrace{t(switch, c_\prec)}^{\text{Switch time}} + \overbrace{t(r,c)}^{\text{Compute time}} + \overbrace{\frac{p(r,c) - p(r,c_\prec)}{p(r,c)} * d(c_\prec, c)}^{t_{echo}} \tag{3.5}$$

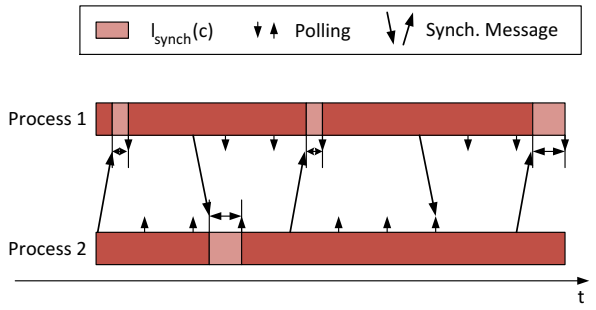
$$\begin{aligned} E(r, c_\prec, c) &= E(switch, c_\prec) + E_{\text{while } c_\prec}(r) + E_{\text{while } c}(r) \\ E(switch, c_\prec) &= P(switch, c_\prec) * t(switch, c_\prec) \\ E_{\text{while } c_\prec}(r) &= P(r, c_\prec) * d(c_\prec, c) \\ E_{\text{while } c}(r) &= P(r,c) * (t(r,c) + (\frac{p(r,c) - p(r,c_\prec)}{p(r,c)} - 1) * d(c_\prec, c)) \end{aligned} \tag{3.6}$$

If the region depends on an external signal (e.g., a synchronization message), the number of instructions $I(r)$ is not constant anymore. In these cases, Equation 3.4 is not applicable but has to be determined based on the used power saving mechanisms. Here, a polling latency $l_{sync}(c)$ determines to which extend the synchronization is prolonged. Such a polling latency is introduced when the optimized configuration disables a component for a certain time and only enables it sporadically to check for the synchronization message. This would be the case with clock modulation, where processor cores are disabled for a certain time period, or a mechanism that replaces the synchronization busy-wait loop with a loop that includes sleep phases. Figure 3.4a depicts such a synchronization, which consists of an initialization phase, the actual synchronization and a finalization phase. In this figure, two scenarios are described with the same polling latency but different throughputs. Configuration 1 provides a higher throughput than configuration 2. Thus, the initialization phase is executed faster. The saved time is spent waiting for the external signal to arrive. Thus, time and energy needed to wait for the synchronization message is increased.

More complex patterns, where multiple synchronization messages are sent and received, increase the overall runtime of the synchronization additionally, as every single message can be delayed. However, this depends on the internal mechanisms of the synchronization library. An example is depicted in Figure 3.4b. Here two processes need five messages for a single synchronization. In average, each of the



(a) two different synchronization scenarios with a common polling latency and different throughputs



(b) synchronization mechanism with five synchronization messages. Depending on the polling latency $l_{sync}(c)$ the runtime is increased.

Figure 3.4: Influence of power saving mechanisms on synchronizations, the initialization phase of the synchronization is marked green or blue (depending on the configuration), the actual synchronization is marked red

messages is delayed by half the polling frequency.

3.1 Structured Programs

A typical program in HPC is structured, e.g., one iteration in the code follows another. Such an iteration can, for example, be one step of a solver or a single time step in a discrete-event simulation [Rob04, Section 2.2.2]. Here, multiple regions that have the same behavior can be defined as a function $f \in F(S)$. $F(S)$ could be all procedures or functions of a software S. A single function definition can also provide a context (e.g., call stack information) to distinguish between different calls within the program. With a given mapping from regions to functions $r \mapsto f$ (e.g., by the function name), the complexity of the given model can be reduced significantly if the number of functions is lower than the number of regions $|F| \ll |R|$. However, functions cannot be directly translated to regions without additional information, e.g., the specific call number n of the function. With the introduction of such information, a mapping of $m(f, n) \mapsto r$ is possible. The principle of functions decreases the amount of data needed for the model. A light weight profiling can be used instead of tracing to create the throughput and power information for each function. However, profiling is not able to record the sequence of events that are processed, and while regions have a specific temporal order, functions cannot be sorted in this way. But since the order of the single regions is important for the model, profiling cannot be used solely to capture all necessary data that the described model needs. Still, tracing is not without any alternative. The temporal relations that are missed with profiling can also be captured and described by event flow graphs (EFGs) [AFL14]. These describe transition rules $x = (f, f_{\succ}, condition) \in X(S)$ of a program S. Each of the transition rules $(f_1, f_2, condition)$ describes a transition from one function f_1 to another f_2: $\forall f_1, f_2 \in F(S) : f_1 \prec f_2 \Leftrightarrow \exists (f_1, f_2, *) \in X(S)$. In Table 3.1, I describe the memory complexity of the captured data for tracing, profiling and event flow graphs. In Table 3.2, I describe the notation on edges of the event flow graphs that I use in this work.
In the following paragraph, I describe the memory complexity of traces, profiles, and EFGs for the small

Measurement tool	Profiling	Tracing	event flow graphs
Memory complexity of performance and power information	$O(\|F\|)$	$O(\|R\|)$	n/a
Memory complexity of chronology	n/a	$O(\|R\|)$	$O(\|X\|)$ +$O(\|F\|)$

Table 3.1: Memory complexity of captured information in different measurement approaches

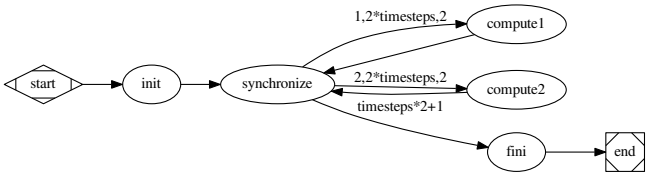
Label	Explanation
(no label)	This transition is executed always
Single number n	This transition is taken when the source node is executed the nth time (1-indexed)
Three numbers i, j, k	This transition is taken when the source node is executed the ith, the $i + k$th time, the $i + 2k$th time, ... until the jth time.
Five numbers i, j, k, l, m	In addition to the three number case, an outer loop is introduced. The source has been executed the nth time, where $n = i + l * y + x * k$ and $i + x * k <= j$ and $1 < y <= m$
Three dots "..."	No pattern could be determined automatically for the successor description

Table 3.2: Notation of event flow graph transition rules (edges)

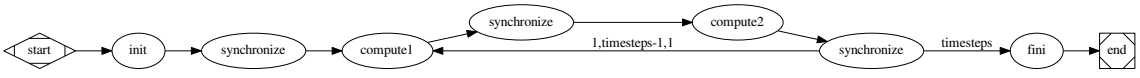
Algorithm 3.1 Example program

```
procedure MAIN(args, argv)
    (data, timesteps) ← init()              ▷ Initialize, timesteps ≥ 1
    synchronize(data)                       ▷ Synchronize data with other ranks
    for i=0;i<timesteps;i++ do              ▷ Iterate the data over several time steps
        compute1(data)                      ▷ Compute next time step phase 1
        synchronize(data)                   ▷ Synchronize data with other ranks
        compute2(data)                      ▷ Compute next time step phase 2
        synchronize(data)                   ▷ Synchronize data with other ranks
    end for
    fini(data)                              ▷ Finalize
end procedure
```

example program that is listed in Algorithm 3.1. A trace of this algorithm will hold ($timesteps * 4 + 3$) entries for performance information about the program structure. This also reflects the chronology, since the entries are stored in order. A profile holds performance information for the five functions init, synchronize, compute1, compute2, and fini. If the profile also includes contextual information about the function calls, it will have two additional entries, since the synchronize function is called in three different contexts. An EFG holds the same number of function definitions and in addition seven, resp. nine transition rules which are depicted as edges in Figure 3.5. However, the given example is simple. The event flow graph of two real applications are depicted in Figure B.1 and Figure B.2 on page 157 ff.



(a) Contex free function scheme



(b) Function scheme with context

Figure 3.5: EFG of Algorithm 3.1. Nodes represent functions, edges depict transition rules.

Algorithm 3.2 Calculation of $\breve{c}$, which is the worst configuration (depending on λ) that a function can have

function CALCULATE $\breve{c}$ FOR ALL f(functions $F(S)$, functions with a prepended tuning step $F_t(S)$, initial function f_0, transition rules $X(S)$, function that maps configurations to tuned functions $c(f)$, evaluation function λ)
  for $f \in F(S)$ **do** ▷ Initialize possible settings
    $C_{possible}(f) \leftarrow \emptyset$
  end for
  $F_t(S) \leftarrow F_t(S) \cup \{f_{-1}\}$ ▷ Register an initial step ...
  $c(f_{-1}) \leftarrow c_0$ ▷ ... with the default configuration that ...
  $X(S) \leftarrow X(S) \cup (f_{-1}, f_0, \text{first call})$ ▷ ... defines the initial setting for f_0
  for $f_t \in F_t(S)$ **do** ▷ add configuration $c(f_t)$ to all possible successors of f_t
    $C_{possible}(f_t) \leftarrow \{c(f_t)\}$ ▷ f_t has only one possible configuration
    $F_p \leftarrow F_t(S)$ ▷ Functions that do not have to be processed anymoreF_p
    $F_\succ \leftarrow \{f_\succ \in F(S) | (f_t, f_\succ, *) \in X(S), f_\succ \notin F_p\}$ ▷ all successors of f_t that are not tuned
    for $f_\succ \in F_\succ$ **do**
      $C_{possible}(f_\succ) \leftarrow C_{possible}(f_\succ) \cup \{c(f_t)\}$ ▷ Add $c(f_t)$ to possible configurations of $f_\succ$
      $F_p \leftarrow F_p \cup \{f_\succ\}$ ▷ Mark $f_\succ$ as processed
      $F_\succ \leftarrow F_\succ \cup \{f_{\succ\succ} \in F(S) | (f_\succ, f_{\succ\succ}, *) \in X(S), f_{\succ\succ} \notin F_p\}$ ▷ Add all successors of $f_\succ$ that are not already processed or tuned
    end for
  end for
  for $f \in F(S)$ **do** ▷ Choose the worst configuration depending on the evaluation function λ
    $\breve{c}(f) \leftarrow \arg\max_{c \in C_{possible}(f)} e(\lambda, f, c)$
  end for
end function

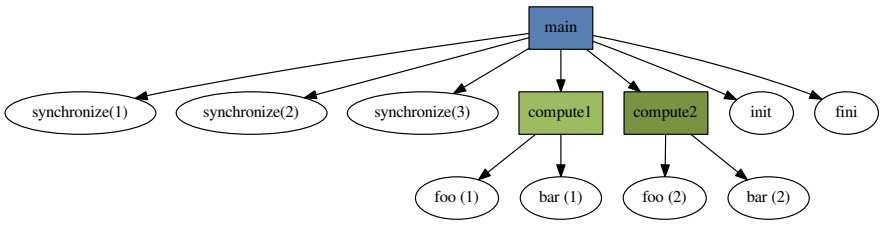
Since one function can represent numerous regions, it is difficult to determine the possible configurations that apply to the regions. This can be seen if the functions *compute1* and *compute2* in Algorithm 3.1 change the configuration to c_1 and c_2, respectively. With a context free processing (Figure 3.5a), the function *synchronize* is executed with three different configurations: c_0, c_1, and c_2. Thus, a simple scaling with the number of function calls and a single configuration is not possible. However, if the possible configurations are known, the worst-case configuration $\breve{c}_\lambda$ can be computed based on an evaluation of the possible configurations of a function. $\breve{c}_\lambda$ is the worst performing configuration (based on λ) that a function can inherit from *any* precessing function. If a new configuration c is applied at the beginning of a function, $\breve{c}_\lambda$ is equal to c. In Algorithm 3.2, I describe how such a worst case configuration $\breve{c}_\lambda(f)$ can be attributed to each function. Based on this worst-case configuration, an upper limit for runtime and energy consumption can be calculated. This is described in Equation 3.7 and Equation 3.8. The upper runtime limit is determined by adding the runtime of all functions at their respective worst case configuration and the time for switching from a previous configuration. Here, the previous configuration with the maximal sum of switching time $t(switch, c_\prec)$ and echo time t_{echo} is considered. The same approach is used for determining the worst case energy consumption, but with the energy consumption of the single runtime components (execution, switching, echo) in mind. Likewise, other metrics like ED^{2}P can be calculated. Furthermore, an upper limit for energy savings of a given configuration can be calculated by applying Algorithm 3.2.

$$t \leq \overbrace{\sum_{f \in F} |\{r \in R | r \mapsto f\}| * t(f, \breve{c}_\lambda(f))}^{\text{Execution time at worst case configuration = (number of occurences of function) x (maximal possible time)}} + \overbrace{\sum_{f \in F | (f_\prec, f, *) \in X(S)} |\{r \in R | r \mapsto f\}| * \max_{c_\prec \in C_{possible}(f_\prec)} \left(t(switch, c_\prec) + \overbrace{\frac{p(f, \breve{c}_\lambda(f)) - p(f, c_\prec)}{p(f, \breve{c}_\lambda(f))} * d(c_\prec, \breve{c}_\lambda(f))}^{\dots t_{echo} \text{ depending on possible previous configurations}} \right)}^{\text{if there is a configuration switch before the function, maximize } t(switch) \text{ and } \dots} \tag{3.7}$$

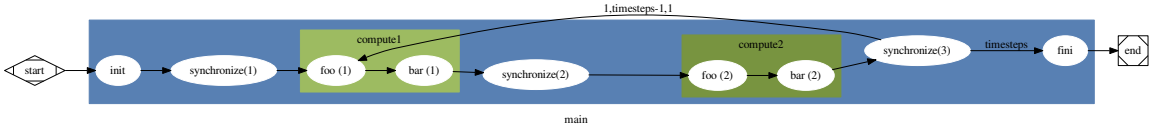
$$E \leq \overbrace{\sum_{f \in F} |\{r \in R | r \mapsto f\}| * t(f, \breve{c}_\lambda(f)) * P(f, \breve{c}_\lambda(f))}^{\text{Energy consumption at worst case configuration}} + \overbrace{\sum_{f \in F | (f_\prec, f, *) \in X(S)} |\{r \in R | r \mapsto f\}| * \max_{c_\prec \in C_{possible}(f_\prec)} \left(\overbrace{t(switch, (c_\prec, \breve{c}_\lambda(f))) * P(switch, c_\prec)}^{\dots \text{worst case switching energy and } \dots} + \overbrace{d(c_\prec, \breve{c}_\lambda(f)) * P(f, c_\prec) + (\frac{p(f, \breve{c}_\lambda(f)) - p(f, c_\prec)}{p(f, \breve{c}_\lambda(f))} - 1) * d(c_\prec, \breve{c}_\lambda(f)) * P(f, \breve{c}_\lambda(f))}^{\dots \text{worst case } E_{echo}} \right)}^{\text{For all called tuning functions find } \dots} \tag{3.8}$$

3.2 Structured Programs with Nested Calls

In the previous section, I described a workflow for exclusive functions, where each function succeeds another. This can be used to describe parallel regions in OpenMP parallel programs that are not nested. However, programs are usually more complex, and functions are able to call each other. This is, for example, covered by call path profiling [KRaM+12]. Here, the functions do not have to be exclusive, but are distinguished according to their call stack, and span a tree where every node represents the execution of a function within a specific context. The nodes in this tree can be differentiated into internal nodes, which represent functions that call other functions and external nodes (leaves), which do not call functions. A simple event flow graph as described in the previous section can only depict nodes of this tree that do not share a common path to its root, since this would contradict the assumption that one function follows another. However, event flow graphs that only hold leaves as nodes are possible. If, for example, the functions `compute1` and `compute2` in Algorithm 3.1 would call the two functions `foo` and `bar`, `compute1` and `compute2` cannot be described in the same EFG, as the representation does not support nested calls. Leaf-Nodes of the call hierarchy, however, exclude each other and can be depicted in a leaf-node-EFG. The call tree and the leaf-node-EFG of the example are depicted in Figure 3.6. However, every function can be the target of an optimization, not only leaves of a calling tree. Thus, the leaf-node-EFGs cannot be used. Alternatively, a new function is assigned to every edge of the EFG. Furthermore, whenever the edge crosses the border of an internal tree node, the newly introduced function is split. Thus, the number of functions and transition rules increases by the number of transition rules and twice the number of internal nodes. Still, the number of functions is lower than the number of regions. I call the resulting graph a complete nested functions event flow graph (CNF-EFG). With such a graph, it is possible to apply the optimizations described for exclusive functions in the previous section also to nested calls. An excerpt of the CNF-EFG based on the leaf-node-EFG from Figure 3.6b is depicted in Figure 3.7. Here, the first blue dots from the left can be interpreted as a function which represents the part of main that is executed before calling init.



(a) Call Tree



(b) Leaf-Node-EFG, internal tree nodes marked as boxes

Figure 3.6: Call Tree and Leaf-Node-EFG for Algorithm 3.1 when compute1 and compute2 call sub-functions foo and bar

Figure 3.7: Excerpt of the CNF-EFG that results from Figure 3.6b. Newly introduced nodes are color coded depending on the internal node of the call tree.

3.3 Determining Efficient Configurations

In order to find efficient configurations for applications, users (1) need to choose an evaluation function λ, which could, for example, be ETS EDP, or ED^{2}P. They (2) need to determine the system parameters d, $t(switch, c)$, and $P(switch, c)$ as well as the application parameters X, F, $p(f, c)$, $t(f, c)$, and $P(f, c)$. These steps are straight forward, given suitable measurement infrastructures. However, based on the parameters a suitable configuration has to be found. Therefore, I propose an algorithm targeted at finding tuned configurations. In a first step, functions that are applicable for tuning are identified. Such functions should have an energy saving potential that is high enough to cover a possible overhead for resetting the applied configuration afterwards. The resetting overhead can be determined from the given successors $f_\succ | (f, f_\succ, *) \in X(S)$ and their parameters. This initial step is described by Algorithm 3.3 and is based on a given default configuration c_0, e.g., the reference setting or the static optimal configuration for the given application.

If there are any functions where a different configuration is applicable ($C_{tuned} \neq \emptyset$), the efficiency in terms of λ increases. This algorithm expects heavyweight functions where an efficiency gain can cover for all efficiency losses due to switching overhead and prolongation. If there are no heavyweight functions, subsequent lightweight functions that are not applicable for tuning can be joined. For such a consolidation, the preceding function should have no other successor and the succeeding function should have no other predecessor. $f' = (f, f_\succ) | \exists!(f, f_\succ, *) \in X(S)$. This procedure can be repeated to create even larger merged functions. Likewise, loops within the EFG can be consolidated if they have only one preceding and one succeeding function $f' = (f, f_\succ, f_{\succ\succ}) | \exists!(f, f_\succ, *) \in X(S) \wedge \exists(f_\succ, f_2, *) \in X(S) : f_2 = f_\succ \vee f_2 = f_{\succ\succ}$. If the energy saving potential of such a merged function is high enough, it can be treated like any other function that is applicable for tuning. If such a consolidation cannot be applied, the

Algorithm 3.3 Finding suitable tuning configurations, initial step

function FIND TUNING CONFIGURATIONS(Functions $F(S)$, Transition Rules $X(S)$, evaluation function λ)
  $C_{tuned} = \emptyset$
  for $f \in F(S)$ **do** ▷ Find saving potential for all functions
    $c_{opt} = c_0$ ▷ Default configuration: c_0
    $P_{opt} = P(f, c_0)$
    $t_{opt} = t(f, c_0)$
    for $c \in C$ **do** ▷ Test fo all c whether switching and prolongation overhead can be covered
      $E_c \leftarrow E(switch, c_0, c) + E(f, c_0, c) + E(switch, c, c_0) + \max_{\forall f_\succ : (f, f_\succ, *) \in X(S)} E_{echo}(f_\succ, c, c_0)$
      $t_c \leftarrow t(switch, c_0, c) + t(f, c_0, c) + t(switch, c, c_0) + \max_{\forall f_\succ : (f, f_\succ, *) \in X(S)} t_{echo}(f_\succ, c, c_0)$[1]
      $P_c \leftarrow \frac{E_c}{t_c}$
      if $f_\lambda(P_c, t_c) < f_\lambda(P_{opt}, t_{opt})$ **then** ▷ if this is more efficient
        $c_{opt} \leftarrow c$ ▷ c is new best configuration
        $P_{opt} \leftarrow P_c$
        $t_{opt} \leftarrow t_c$
      end if
    end for
    if $c_{opt} \neq c_0$ **then**
      $C_{tuned} \leftarrow C_{tuned} \cup \{(f, c_{opt})\}$ ▷ Store best configuration
    end if
  end for
  for $(f, c), (f_\succ, c_\succ) \in C_{tuned} | (f, f_\succ, *) \in X(S)$ **do** [2]
    $t_c \leftarrow t(switch, c_0, c) + t(f, c_0, c) + t(switch, c, c_\succ) + t_{echo}(f_\succ, c, c_\succ)$
    $E_c \leftarrow E(switch, c_0, c) + E(f, c_0, c) + E(switch, c, c_\succ) + E_{echo}(f_\succ, c, c_\succ)$
    $P_c \leftarrow \frac{E_c}{t_c}$
    if $f_\lambda(P_c, t_c) > f_\lambda(P(f, c_0), t(f, c_0))$ **then** ▷ if the default is more efficient, remove predecessor from tuning configurations
      $C_{tuned} = \{(f_x, *) \in C_{tuned} | f_x \neq f\}$
    end if
  end for
  return C_{tuned}
end function

remaining lightweight functions cannot be tuned. However, all subtree functions beneath a given node of the call-tree can also be combined. For example, if all functions under main were merged, only one function would remain. Under such circumstances, Algorithm 3.3 would determine the static optimal configuration for the application. The algorithm only finds suitable configurations if all overheads from setting, resetting, and prolongation can be covered with the efficiency gain. Thus, it will not necessarily find good configurations when two subsequent functions have configurations that differ from c_0. Still,

[2]Please note that the worst case successor in terms of energy and time can differ. The result of this algorithm is the worst case assumption (highest energy and highest time)

[2]If two follow-up functions are tuned, a reset between these functions is not necessary anymore. Additionally, the successor function is initially executed with c and not c_0. Therefore, t_{echo} and E_{echo} change. Even though it is unlikely that the saving for not resetting c cannot cover for possible efficiency gains, here such cases are filtered. If this happens, the predecessor function is not tuned anymore.

the returned configuration will be at least as efficient as c_0, as described in Equation 3.9.

$$\forall f \in F(S) : e(\lambda, f, c_f) \begin{cases} > 1 \text{ if } \exists (f, c) \in C_{tuned} \\ = 1 \text{ otherwise} \end{cases} \tag{3.9}$$

After the assignment of tuning configurations and a consolidation of functions, non-tuned functions that precede or succeed a tuned function can be combined with tuned functions if the configuration of the tuned function is more efficient than the default. This can be repeated to include neighbors of the merged tuned function. This step also improves the efficiency if there are any neighboring functions that can be included. If not, the efficiency does not change. This is described for a preceding non-tuned function in Equation 3.10

$$\forall f \in F(S) | \exists ! (f, f_{\succ}, *) \in X(S), (f_{\succ}, c_{\succ}) \in C_{tuned} : e(\lambda, f, c_f) \begin{cases} > 1 \text{ if } e(\lambda, f, c_{\succ}) > e(\lambda, f, c_0) \\ = 1 \text{ otherwise} \end{cases} \tag{3.10}$$

In the next step, configuration resets are removed if a new configuration is set right afterwards. Resets can also be removed if the overhead of the resetting is higher than the efficiency-loss for non-tuned functions that occur between two tuned functions. Furthermore, the setting of configurations can be skipped for every function where all predecessor functions already applied the same configuration. All of these actions decrease the switching overhead and thus increase the efficiency.

The configuration created by this strategy is not necessarily optimal. This could be achieved by re-running several traces of an application in a trace-based simulator. The different traces would be gathered by testing configurations for the individual regions. However, such a global optimum would require to process a much larger search space. While a trace contains information about each region that is executed, my model uses performance and power profiles in addition to an EFG. Therefore, the number of processed items is much smaller, as I summarize in Table 3.1. To further reduce the complexity of such an an optimization approach, one could omit the transition rules as it is done by the READEX project [SGK+17]. However, this would make it impossible to join contiguous functions that use the same configuration. Furthermore, if the transition rules are not clear, the initial configuration of each function would be unknown. Hence, the worst case switching overhead and latency would have to be assumed when determining tuning configurations. The heuristic that I proposed in this section represents a balanced alternative to the trace-based analysis, which could provide a better tuning result but has a high memory and processing complexity, and a pure profile based tuning strategy with a low complexity and a sub-optimal outcome.

3.4 Challenges

The model that I describe in this chapter has several weaknesses. First of all, it relies on power and performance measurements for single regions or functions. At the beginning of my thesis, there was no tool that was able to capture the needed parameters. In Section 6.2, I describe how I extended VampirTrace to be able to capture power data in addition to performance data. Another challenge is the granularity of power measurements. Power meters are not able to measure the power consumption of a single core, since the voltage input of the different processor cores is not separated. Thus, measured power values cannot be mapped to a single region. One way to overcome this limitation is to execute the same function simultaneously on all processor cores, e.g., when a SPMD fork-join paradigm like OpenMP loop-parallelization is used. Another way would be to use performance counter based models. However, such models can be inaccurate. A third solution would be based on a trace that includes power and performance data. This trace could be parsed and power measurements that are taken when a single function is executed on all cores that are covered by the power measurement could be applied to the respective function.

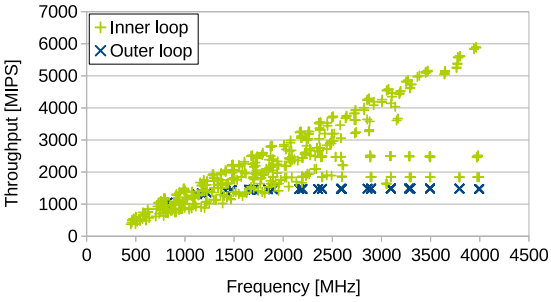
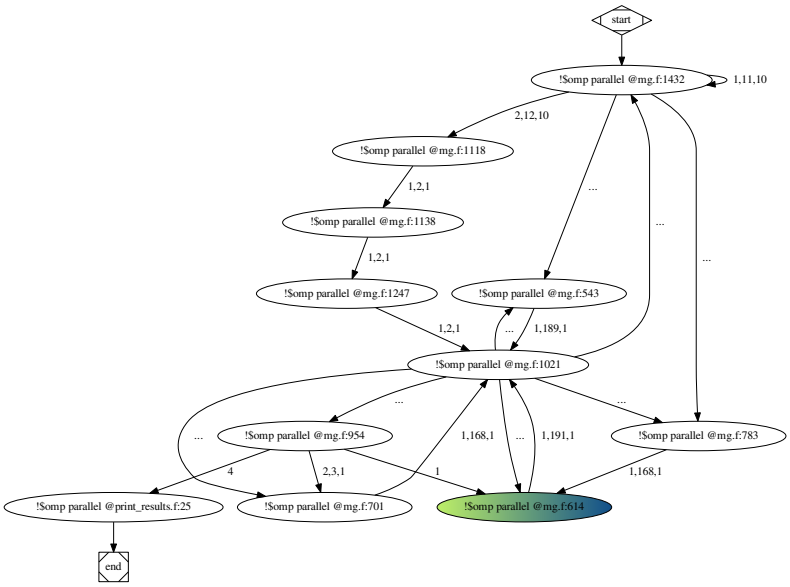
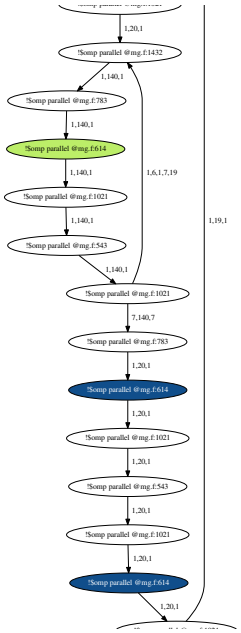


Figure 3.8: Different runtime behavior of the OpenMP parallel region `mg.f:614` from the NAS Parallel Benchmark MG (Class C, Intel Core i7-6700K test system). The runtime behavior depends partially on the context in which the loop is called. If the function is called within the inner loop (Figure 3.9b), the throughput scales with the frequency. If it is called from the outer loop it does not. Calls from the outer loop contribute 95 % to the overall runtime of the function (not depicted).



(a) exclusive EFG without stack information



(b) exclusive EFG with stack information (subset)

Figure 3.9: Exclusive event flow graphs of NAS Parallel Benchmark MG (OpenMP, Class C). The stack information is needed to distinguish OpenMP parallel loops with different call stacks and contexts.

Second, the overhead to distinguish different functions has to be weighed against the benefit of a more exact function description. Figure 3.8 depicts the runtime behavior of the OpenMP parallel region `!$omp parallel @mg.f:614` from the NAS Parallel Benchmark MG in CLASS C on an Intel Core i7-6700K test system. This region makes up more than 50 % of the overall runtime of the program. However, the behavior of the function is split into some regions that scale with the frequency and some regions that do not. When adding call stack information, the differently behaving functions with the same name can be distinguished. However, getting the call stack can significantly influence the measurement and the performance of a tuning based on this information. Figure 3.9 shows excerpts of the exclusive event flow graph of these applications. In Figure 3.9a, the different incarnations of the parallel regions cannot be distinguished. With the addition of call stack information (Figure 3.9b), a differentiation is possible. Still, the question which additional data has to be captured to distinguish regions and map them to functions cannot be answered easily and depends on the given application that is to be tuned. While for one application a function name can be sufficient, another one might need the call stack, or even input

parameters that might be indicating a specific data set size. If the gathering of the additional information is too cost intensive, more coarsely grained functions could be used. This can be done by clustering functions until they are heavy-weight enough for tuning. A maximal function cluster would include all functions of an application. Here, a static optimal configuration would be applied.
A final challenge is the number of input parameters for the model. These can be obtained by an exhaustive search of the parameter space. However, such an approach is time consuming and increases potentially with each parameter that is added to the configuration space. Interpolations can be used to estimate the power consumption and performance implications of specific configurations. For example, in Figure 7.4, not all uncore frequencies are tested, but an interpolation is utilized to compute the energy efficiency.

3.5 Conclusion

In this section, I described a model that can be used to determine tuning configurations for different functions within a program. This model needs various input parameters. First of all, system information that describes parameters of power saving features has to be determined. This includes information about switching latencies for changing the respective feature, access costs for such a switch, and a general idea on the influence of power consumption and performance when they are applied. These parameters are discussed in Chapter 4. Additionally, information about the software is needed. To limit the switching overhead, subsequent functions can be joined. However, such an optimization needs information about the internal structure of the software. The interaction of hardware and software determines the performance and power consumption of single functions on different platforms. This, however, would need a tool that unifies performance and power measurements, which is discussed in Chapter 5 and implemented in Section 6.2. In the last step, the tuning measures have to be applied. This includes providing access to the respective hardware features for user-space applications (discussed in Section 6.1) as well as the possibility to interrupt the software at specific points in time (discussed in Section 6.3).

4 Hardware Model Parameters

We are stuck with technology when what we really want is just stuff that works.
The Salmon of Doubt by Douglas Adams

In this chapter, I describe how the hardware dependent parameters of the model that I introduced in Chapter 3 can be measured. I start this chapter with a description of how the executed workload influences the power consumption of computing systems, which inherently defines the power saving potential. Later on, I lay out algorithms to measure the switching overhead $t(switch, c)$ and the (de-)activation latency d for the implemented ACPI based power saving mechanisms. In addition, I present results for the test systems listed in Appendix A and describe how the usage of power saving mechanisms influences the throughput of memory bound workloads. This can be considered to be the lower limit for the performance impact on applications as discussed in Section 4.2. Section 4.3 covers methods and results for P-state model parameters, Section 4.4 and Section 4.5 focus on C-states and T-states, respectively. Some of the presented results have been previously published in [SHM12, SMW14, HSI+15] and [SIB+16]. I conclude this chapter with a summarization of the findings in Section 4.6.

4.1 Impact of Power Saving Mechanisms on Power Consumption

The power saving potential of different power saving mechanisms depends on the mechanisms' scope and the characteristics of the executed code regions and can thus not be validated by a single benchmark. In this section, I describe how the application of P-states affects the power consumption of active cores. However, the findings can also be applied to other mechanisms like C-states and T-states.
The power impact of power saving mechanisms can be estimated by using Equation 2.1. The reduction of the frequency reduces the dynamic power consumption of a processor linearly. A lower voltage decreases dynamic and static power consumption. However, this only applies to the parts of a processor that are actually affected by the new P-state. This can vary between architectures. For example, on the Westmere-EP and the AMD test systems, the uncore and its portion of the processor power dissipation is not affected at all. On the Sandy Bridge and Ivy Bridge Systems, cores and uncores share a common voltage and frequency. On Haswell and Skylake systems, the uncore frequency is regulated by hardware and the core frequencies influence this decision. Thus, the actual effect on the processor power consumption needs further investigation.
A lower limit for power reductions via DVFS can be determined by comparing the C1 states with different frequencies. Here, the cores are still active, but the activity of the cores and consequently the dynamic power consumption of the processor is reduced to 0. Even though the power consumption under a specific C-state seems to be an unconnected problem to the power impact of P-states, it marks a valid lower bound. Instructions with a high latency, like `pause`, `sqrt`, and `div`, result in resource stalls that also reduce the activity factor and the power consumption of a processor. For example, the `vdivpd` instruction on Broadwell processors has a latency of 25-35 cycles [Int16a, Table C-8]. If the out-of-order engine is unable to fetch more instructions due to data dependencies, most of the processor waits for this time. In a region where one of these instructions follows another, this can lead to a situation where every 25th to every 35th cycle the processor executes a new instruction, which represents $\approx 3\,\%$ of the overall cycles. C1 can be compared to a situation where this is reduced to $0\,\%$. In [MHSM10], Molka et al. use the `sqrtpd` instruction to create a baseline in order to determine costs for executing instructions and data transfers. On the systems they investigated, this instruction has a latency of up to 30 cycles (Westmere-EP based Intel Xeon) and 27 cycles (Istanbul based AMD Opteron). In workloads where the processor cores stall significantly, there is little potential for power savings.

Algorithm 4.1 Benchmark for determining the possible range for P-state power savings

```
...                                                    ▷ disable C-states above C1
for f in available frequencies do                      ▷ Run benchmarks for all frequencies
    t(f, start) ← get_time()
    ./FIRESTARTER -t 300                               ▷ Run FIRESTARTER for 5 minutes
    t(f, med) ← get_time()
    sleep 300                                          ▷ idle for 5 minutes
    t(f, end) ← get_time()
end for
for f in available frequencies do          ▷ Get power data and compute potential per frequency
    P_firestarter(f) ← max(P_sample(t) ∈ (sampled power values)|t(f, start) < t < t(f, med))
    P_c1(f) ← min(P_sample(t) ∈ (sampled power values)|t(f, med) < t < t(f, end))
end for
```

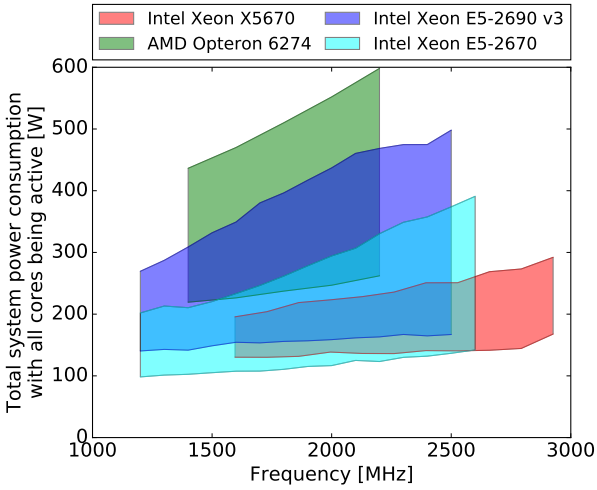


Figure 4.1: Power consumption of x86 based server systems for different P-states when all cores are active. The activity factor of the processor when executing a given workload and the applied frequency determine the power consumption. A minimal activity factor is given with the C1 idle state. Processor stress tests like FIRESTARTER cause a maximal activity factor. The activity factor and the power consumption of applications range between these extrema.

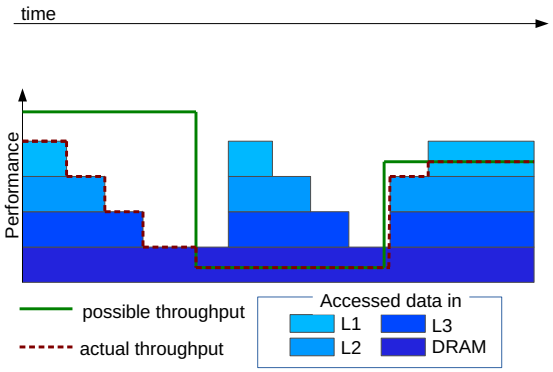
The upper limit for P-state power-savings can be determined by increasing the P-state while running an application that maximizes the power consumption of a processor. In [HOMS13], Hackenberg et al. present FIRESTARTER[1], which is a tool that is targeted at maximizing the dynamic power consumption of x86 processors. The dynamic power consumption increases with frequency, voltage, and the activity factor. FIRESTARTER targets the latter by stressing the SIMD computing resources and loading data from different memory levels to the processor cores. Hackenberg et al. showed in [HOMS13, HSI+15] that their approach is superior to other processor stress tests in achieving this goal.

I measure the two different benchmarks for all available frequencies on the four server systems listed in Appendix A. The test script contains a loop that alternately executes FIRESTARTER and keeps all processor cores in C1. Each of these phases is executed for five minutes. Afterwards, the maximal power consumption of the FIRESTARTER phase and the minimal power consumption of the idle phase is collected. While the benchmarking process is described in Algorithm 4.1, Obtained results for the different test systems are depicted in Figure 4.1. These indicate that the amount of power saving that can be achieved with P-states significantly depends on the workload. Thus, if these are about to be tuned for energy efficiency, the power consumption of the system must be monitored or modeled to gain information about their efficiency. Furthermore, the power consumption of the other components within the compute node limits the effectiveness of processor power saving mechanisms. If the optimized scope only represents a smaller part of the overall power consumption, the saving potential in relation to the node power is limited.

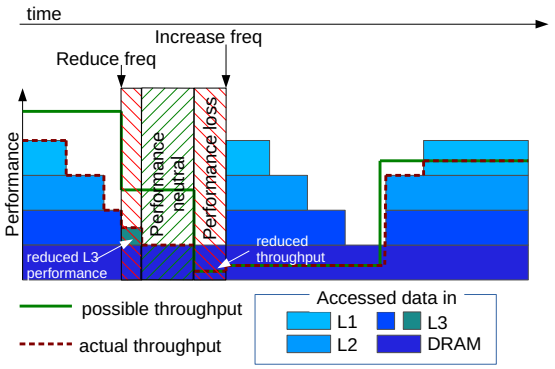
[1] https://github.com/tud-zih-energy/FIRESTARTER

4.2 Impact on Performance

I layed out the upper limit for performance impacts in Section 2.6 and summarized them in Table 2.3. In short, each reduction in frequency lowers the performance linearly. Thus, when the average core or uncore frequency is reduced via P-states, T-states, and C-states, the resulting performance matches the average frequency over the whole time period. However, this only represents the worst case performance loss and does not cover all possible workloads. It is intuitive to say that the performance of an application, which uses only core resources, is not reduced when the uncore frequency is lowered or unused resources are disabled. Thus, a reflection on a lower limit can improve the predicted performance impact.
The roofline model [WWP09] by Williams et al. defines that the performance is always limited by a specific resource. The performance of a non-limiting resource can, by definition, be throttled without affecting the overall performance. E.g., if a processor can add eight double precision values per cycle, but is only able to load 16 bytes, then the processing width could be limited to two values per cycle without hurting performance. Alternatively, the floating point units could be slowed down by a factor of four. If the limiting resource is not influenced by a power optimization, the performance of a program will stay the same as long as the optimized resource does not become the new bottleneck. Despite the popularity of the roofline model, its applicability to existing processors and workloads is limited, since it *"cannot describe relevant bottlenecks beyond memory bandwidth and peak performance"*, as described by Stengel et al. [STHW15]. An alternative approach [TH10] is introduced by Treibig and Hager. They describe the memory bottleneck as a result of compute capabilities and possible bandwidths to different memory levels. Based on the location of the accessed data in the memory hierarchy, their Execution-Cache-Memory (ECM) model can be used to model the throughput of computing routines. This information can be used for energy efficiency optimizations [HF16]. In Figure 4.2, I detail how a DVFS optimization can introduce new bottlenecks. In this thesis, I assume a workload where the memory bandwidth is the limiting resource as the lower limit for performance impacts.



(a) Default configuration. First performance is limited by data accesses, then by instruction latencies, then by data accesses, and finally by instruction latencies.



(b) Application of DVFS to the program. In this example, it is assumed that the L3 cache and the processor cores share a frequency. Thus, if the processor frequency is lowered, not only the processing performance is reduced but also the throughput of the accessed memory level (L3). When DRAM becomes the bottleneck, the processing performance is not influenced. When instruction latencies become the bottleneck, the application of DVFS reduces the throughput.

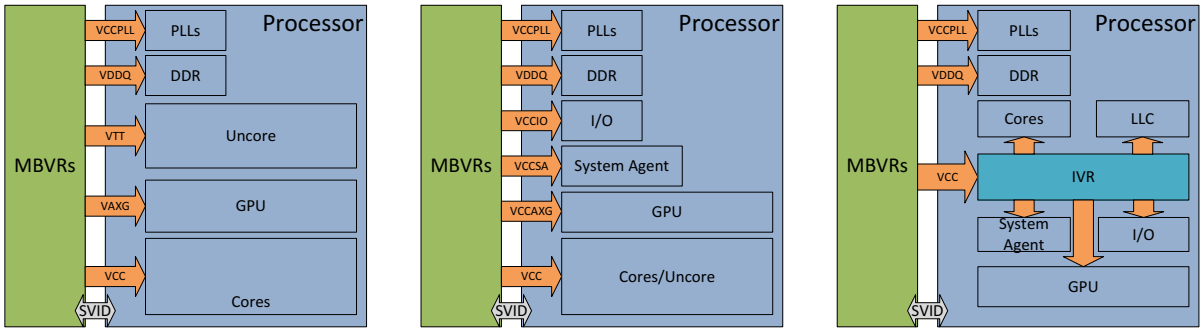
Figure 4.2: The performance of a program (red dotted line) is limited by data location (blue boxes) and processor throughput for the given instruction mix (green line). For long running instructions (like `fsqrt`), the processor throughput can be lower than the limit that is inflicted by the DRAM bandwidth.

4.3 ACPI P-States

I introduced P-states and their implementation technique DVFS in Section 2.6. In this section, I describe parameters of P-state changes and the influence of P-states on the performance of memory bound workloads.

4.3.1 Scope

While the frequency of a processor can be changed easily, a voltage change needs the support from voltage regulators (VRs), which can be located in the processor or on the mainboard. To distinguish these classes, I use the terms integrated voltage regulator (IVR) and main board voltage regulator (MBVR) based on Intel's naming scheme given in [Int15b]. A VR has two major functions in providing voltage to a computing device: to stabilize the supplied voltage and to change the supplied voltage according to the needs of the device. To change the supplied voltage, the processor has to instruct the voltage regulators via a common interface. This interface is called Serial Voltage Identification (SVID) [Int15b, Section 2.2.9] by Intel and Serial VID Interface (SVI) [Adv13, Section 2.5.2.1] for AMD processors. The number of MBVRs defines the number of different voltage domains that a processor can have. If the processor supports IVRs, the number of different voltages increases accordingly. Voltage regulators and the attached processor resources are exemplarily depicted for three desktop processors in Figure 4.3. A set of components that is attached to a common supply voltage is called voltage domain. Components within such a voltage domain cannot regulate their voltage independently from each other. However, their frequency can still differ. This means that within a voltage domain, only DFS can be used for fine grained P-states as it is, for instance, implemented in certain AMD processors. P-state domains of the used processors are listed in Table 4.1. When multiple cores share a single P-state domain but have different settings, the lowest P-state (i.e., the highest frequency) of all active cores (i.e., all cores that do not reside in a deep C-state), is applied. If a processor supports Hyper Threading Technology, the P-state of a core is defined by the highest frequency of its hardware threads.



(a) Intel Nehalem HE [Int11c] (b) Intel Sandy Bridge HE [Int13b] (c) Intel Haswell HE [Int14a]

Figure 4.3: Schematic illustration of voltage regulators and the attached components for three desktop processors

Table 4.1: P-state and other DVFS domains of different x86 processors

Architecture	P-state domain	Additional DVFS domains
Intel Nehalem/Westmere HE/EP	all cores: DVFS	uncore
Intel Sandy Bridge/Ivy Bridge HE/EP	all cores and uncore: DVFS	GPU (if avail.)
Intel Haswell/Broadwell/Skylake HE	all cores: DVFS	uncore, GPU
Intel Haswell/Broadwell EP	each core: DVFS	uncore
AMD Family 15h	each module DFS, all modules DVFS	uncore, GPU and memory (e.g., models 10h-1Fh)

4.3.2 Access Costs $t(switch, c)$

Linux provides two different interfaces to change processor frequencies. The standardized `cpufreq` interface [PS06] that works on all architectures, and the x86-specific `msr` kernel module that provides a direct access to the hardware interface. In addition, the library `libcpufreq`, which is part of the Linux kernel's user tools is often used to write software that accesses the `cpufreq` interface. In this section, I provide performance results that describe the access overhead when using one of these features to change processor P-states.

I use the most efficient way to access an interface. This means that I open possible file descriptors at an initialization stage and set-up the data that is written to the interface. In the measurement phase, the algorithm just writes the prepared data to the file descriptors. I run the benchmark as a privileged user to avoid the restrictions of the `msr` kernel module. The task that issues the change request is also pinned to the hardware thread whose frequency setting is changed. To filter out outliers, I run each benchmark 10 times and use the median of the results. Table 4.2 lists results from three of the analyzed test systems. On all systems, the standardized access via `libcpufreq` is multiple orders of magnitude slower than the others. This can be attributed to the internal handling of the user space library that opens, handles, and closes a number of files of the *cpufreq* interface. A direct access to the respective sysfs files has the lowest runtime among the tested procedures on all examined systems, due to the internal handling of the request. The write to the respective register is held back by the operating system if the register setting does not change. If the frequency is actually about to change, the respective access time increases. To measure such an overhead, an actual frequency transition would have to be executed. If this is done, the cpufreq ACPI driver, which is used for AMD and Intel processors, reads and writes the respective MSRs, which increases the switching time. For example, on the Intel Xeon E5-2670 and Intel Xeon E5-2680 v3 test systems, the access costs of one switch increases to 1.74 and 2.34 µs, respectively when the frequency is toggled between 2.4 and 2.5 GHz. However, this result cannot be related to a specific configuration. Still, a direct access to the MSR that holds the P-state information is desirable, since it bears the lowest overhead. I describe that the given kernel infrastructure is not designed to provide non-privileged users with such a direct access and provide an alternative mechanism in Section 6.1.

4.3.3 Latency d

In [MLPJ14], Mazouz et al. present FTaLaT, which is a framework to measure P-state transition latencies. These latencies describe the time the processor needs to actually change the frequency after a frequency change is initiated by writing to the respective MSR. In their framework, they measure this latency from user-space by accessing the respective `sysfs` files. Mazouz et al. do not only present

Frequency [GHz]	1.6	1.733	1.867	2.0	2.133	2.267	2.4	2.533	2.67	2.8	2.933
MSR	0.6	0.56	0.54	0.5	0.47	0.46	0.43	0.42	0.4	0.39	0.38
sys/cpufreq	0.62	0.57	0.53	0.5	0.46	0.44	0.41	0.39	0.37	0.36	0.34
libcpufreq	34.7	32.06	29.84	27.85	26.1	24.55	23.12	21.95	20.89	19.88	19.06

(a) Intel Westmere-EP processor Intel Xeon X5670

Frequency [GHz]	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0	2.1	2.2	2.3	2.4	2.5	2.6
MSR	1.78	1.66	1.56	1.47	1.4	1.33	1.27	1.21	1.15	1.12	1.07	1.04	1.0	0.97	0.95
sys/cpufreq	0.82	0.75	0.7	0.65	0.61	0.58	0.54	0.52	0.49	0.47	0.45	0.43	0.41	0.39	0.38
libcpufreq	42.66	39.68	36.59	34.22	32.08	30.25	28.67	27.04	25.93	24.59	23.47	22.42	21.39	20.57	19.8

(b) Intel Sandy Bridge-EP processor Intel Xeon E5-2670

Frequency [GHz]	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0	2.1	2.2	2.3	2.4	2.5
MSR	1.66	1.6	1.55	1.45	1.36	1.28	1.23	1.21	1.15	1.09	1.05	0.99	0.96	0.92
sys/cpufreq	0.7	0.65	0.6	0.56	0.53	0.49	0.47	0.44	0.42	0.4	0.38	0.36	0.35	0.34
libcpufreq	29.62	27.58	25.35	23.79	22.17	20.85	19.66	18.72	17.75	16.79	16.06	15.44	14.73	14.06

(c) Intel Haswell-EP processor Intel Xeon E5-2680 v3

Table 4.2: Access costs for different DVFS interfaces on Intel server processors in µs

Algorithm 4.2 FTaLaT measurement loop [MLPJ14]

```
Require: target_time_interval, target_frequency ▷ Performance must be within the target_time_interval
  cpufreq_setspeed(cpu0,target_frequency)                          ▷ set initial frequency
  init_time ← rdtsc()                  ▷ Start of measurement after accessing the sysfs interface
  repeat                                    ▷ Measure execution time of measurement_loop ...
      start_time ← rdtsc()
      measurement_loop()
      stop_time ← rdtsc()
  until stop_time - start_time ∈ target_time_interval     ▷ ... until target performance is reached
  ...                                                                  ▷ Verify measurement
  if verified then
      measured_time ← stop_time - init_time
  end if
```

their tool, but also show that the transition time depends (1) on the processor generation, (2) the difference between initial and target frequency, and (3) the direction of the transition (i.e., the increase vs. the decrease of the frequency). To do so, FTaLaT measures the runtime of a small workload (measurement_loop) multiple times for the initial and the target frequency. Based on these results, a performance range (target_time_interval) is defined. In the measurement phase, FTaLaT switches the frequency and repeats the measurement loop until its runtime is within the expected range. Algorithm 4.2 lists how a single transition measurement is taken. The measurement is taken on the first hardware thread, while the rest of the system is supposed to be idle. I co-authored a publication from Hackenberg et al. [HSI+15] where I revisited the idea and presented information on a newer Intel Haswell processor.
The results presented in this section have been gathered by running FTaLaT 500 times for each source and target frequency. Afterwards, I determined the trimmed mean, where I filtered out results that deviated more then the triple standard deviation from the mean.

Intel Westmere EP

Intel Westmere EP processors have a common voltage and frequency domain across all cores. According to the technical manual [Int11d, Section 8.5.], a P-state change is implemented as follows:

> *"– If the target frequency is higher than the current frequency, V_{CC} is ramped up in steps by placing new values on the VID [voltage identifier] pins and the PLL [phase-locked loop] then locks to the new frequency.*
>
> *– If the target frequency is lower than the current frequency, the PLL locks to the new frequency and the V_{CC} is changed through the VID pin mechanism."*

Thus, a P-state reduction (increasing the frequency) takes a considerable amount of time for SVID communication, voltage ramp-up and a final PLL frequency change. A P-state increase is not defined precisely, as the manual does not mention whether the frequency is changed before the new voltage is applied or during the voltage change. The results depicted in Figure 4.4 indicate the former. Additionally, measurements confirm that all cores share the same frequency. The actual P-state is defined by the lowest P-state of any core on a processor. Thus, a frequency reduction is only executed when all processor cores agree to a lower frequency. Independently of source and target frequency, a frequency reduction uses approximately 10 µs. As I said before, this however does not mean that the transition is completed as the voltage change is not necesseraily included in the measurement. In contrast, a frequency increase takes at least twice the time when switching to the next available frequency. Depending on the difference between source and target frequency, this latency can increase significantly to up to 64 µs. This is in accordance with the results presented in [MLPJ14, Fig. 3]. The uncore is located in a separate domain. The supply voltages V_{TTA} and V_{TTD} are described as the *"uncore analog voltage"* and the

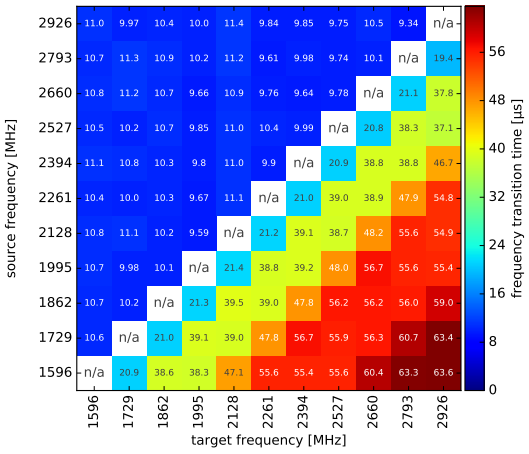


Figure 4.4: Frequency transition delays for Westmere EP based server processor Intel Xeon X5670

"uncore digital voltage", respectively [Int11d, Table 2-7]. Both should be *"derived from the same voltage regulator"*. The actual voltage that is supplied is to be programmed via SVID [Int11d, 2.1.7.1] and influences the frequency range in which the uncore can be operated under stable conditions. Knowing the uncore frequency is crucial in determining the performance of software that is executed on the processor cores. Thus, I measured it via manipulation of MSRs related to Uncore Performance Monitoring Counters (UPMCs). According to measurements of the UPMC `MSR_UNCORE_FIXED_CTR0` [Int15a, Section 18.8.2] under different conditions, the used processor has an uncore frequency of 2.67 GHz, and is independent of the number of used cores and the used core frequency. When all cores use deep ACPI C-states, the counter does not increase with the same rate. However, PMCs only provide statistical information over a time period. Hence, the given average number of uncore cycles does not allow to distinguish whether the uncore stopped for a specific time period, the frequency has been scaled down, or whether both mechanisms have been active.

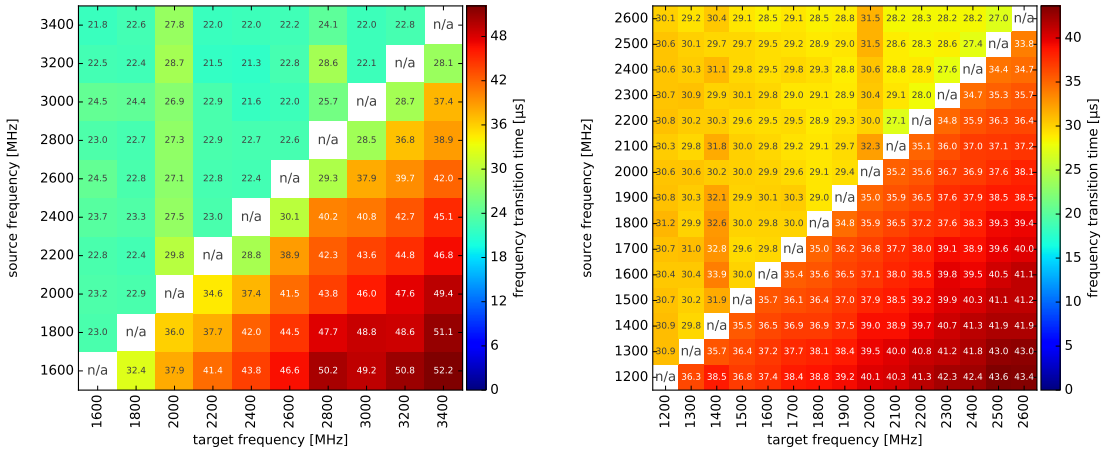
Intel Sandy Bridge

In addition to the description given in the previous section about Westmere processors, the technical documentation for Sandy Bridge processors [Int12b, Section 4.2.1] explicitly describes that all cores share a single P-state:

> *"– All active processor cores share the same frequency and voltage. In a multi-core processor, the highest frequency P-state requested amongst all active cores is selected."*

Thus, the core P-state behavior is the same for Westmere and Sandy Bridge processors. However, the measured P-state latencies depicted in Figure 4.5 have changed significantly in comparison to its predecessor. While the desktop processor has a latency of about 22 µs for frequency reductions, the server processor has a higher latency of about 28-31 µs, depending on the target frequency. When the core frequency is to be increased, the latency also increases depending on source and target frequency up to a maximum of 52.5 µs, and 43.1 µs, respectively. The desktop results are in accordance with [MLPJ14, Fig. 2]. In contrast to its predecessors, Sandy Bridge based processors do not support a separate voltage supply for the uncore, but the core voltage V_{CC} also powers *"lowest level caches (LLC), ring interface, and home agent"* [Int12b, Table 6-16], which represent a significant part of the uncore. According to [RNA+12], desktop processors also use a *"... shared power plane [that] feeds all CPU cores, the ring, and the LLC."* According to measurements with the Linux perf tools[2], the uncore frequency is also determined by the frequency of the processor cores.

[2]Measuring uncore and core cycles via
`sudo perf stat -C 0 -e "uncore_cbox_0/event=0x00,umask=0x00/",cycles sleep 1`



(a) Sandy Bridge based desktop processor Intel Core i7-2600

(b) Sandy Bridge based server processor Intel Xeon E5-2670

Figure 4.5: Frequency transition delays for Intel Sandy Bridge based processors

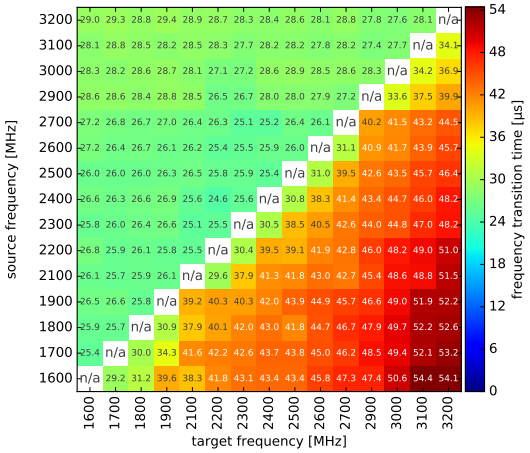


Figure 4.6: Frequency transition delays for Ivy Bridge based desktop processor Intel Core i5-3470

Intel Ivy Bridge

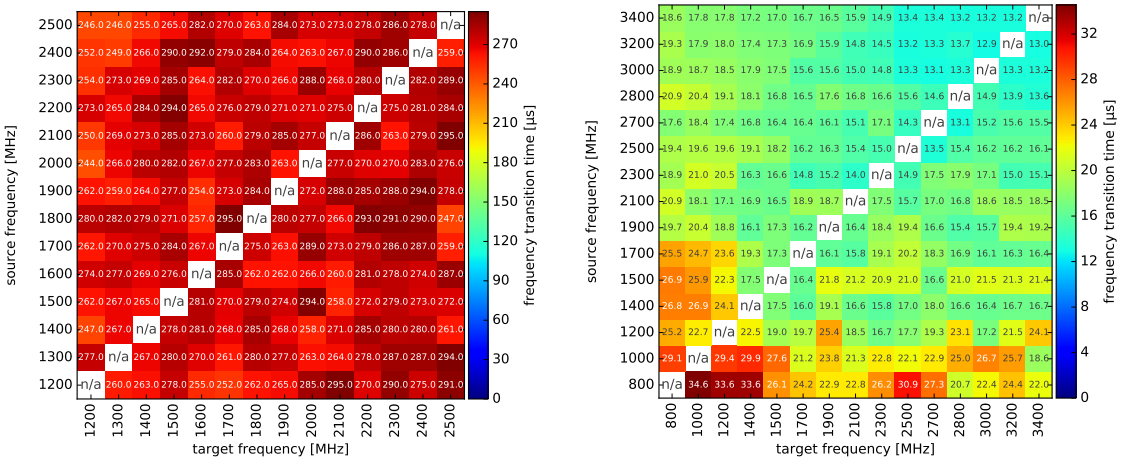
The technical documentation [Int13d] of the Ivy Bridge desktop processor describes no additional improvements or changes in the handling of P-states compared to the Sandy Bridge predecessor. Thus, the absolute values of the results presented in Figure 4.6 do not change significantly. For frequency reductions, the transition times increase with a higher source frequency. This could not be observed on the preceding architecture. For frequency increases, the latency rises with the difference between source and target frequency.

Intel Haswell

The Intel Haswell processor provides a significantly different architecture, as it encapsulates integrated voltage regulators (IVRs). This enables the processor vendor to define more fine grained voltage domains within the processor. Intel defines two features: Uncore Frequency Scaling (UFS) and Per Core P-states (PCPS) that relate to the IVRs. However, the latter is only available for server processors. The technical documentation for desktop parts [Int14a, Section 4.2.1] declares that:

> “ – *All active processor cores share the same frequency and voltage. In a multi-core processor, the highest frequency P-state requested among all active cores is selected.* ”

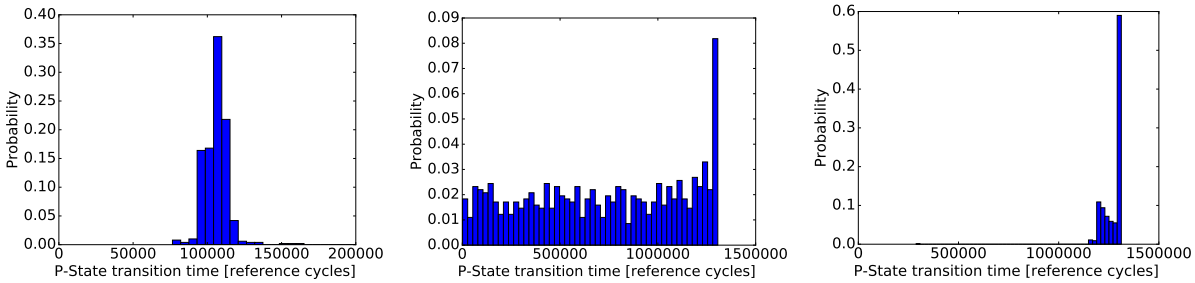
Thus, it can be expected that these behave similar to previous architectures. The initial results are depicted in Figure 4.7. However, none of them provides a clear patterns.



(a) Haswell based server processor Intel Xeon E5-2680 v3

(b) Haswell based desktop processor Intel Core i7-4770

Figure 4.7: Frequency transition delays for Intel Haswell based processors using the original version of FTaLaT



(a) Sandy Bridge based server processor Intel Xeon E5-2670

(b) Haswell based server processor Intel Xeon E5-2680 v3

(c) Haswell based server processor Intel Xeon E5-2680 v3 (fixed version of FtaLaT)

Figure 4.8: Frequency transition delay histograms measuring using FTaLaT for switching from 1.5 GHz to 2.5 GHz

Figure 4.8b depicts the distribution of the results for switches from a frequency of 1.5 GHz to 2.5 GHz on the Intel Xeon E5-2680 v3 test system. When compared to previous Intel server processors (see Figure 4.8a), the distribution is much wider. However, FTaLat does not support the features needed to explain the reason for this behavior. Thus, I changed it in the following way[3]: The frequency check that asserts that the source frequency is set now uses Performance Monitoring Counters instead of reading sysfs files. The user can define a wait time in µs. After the source frequency is set, the algorithm waits for this time before setting the target frequency. Furthermore, I repeat the measurements within one execution of the benchmark where previously a bash script executed multiple iterations of the measurement binary. The new algorithm is described in Algorithm C.2. When applying this optimized version on the Intel Xeon E5-2680 v3 test system with a specific wait-time of 500 µs, the distribution is much more narrow, which is depicted in Figure 4.8c. Based on these measurements, one can conclude that an external mechanism changes the processor frequencies at a regular interval. This is depicted in Figure 4.9. In addition to the 500 µs period from the external source, there is a switching time of about 21 to 24 µs for the actual frequency change.

The desktop processor results also have a different distribution compared to previous generations and

[3]The optimized version of the benchmark is published at `http://github.com/rschoene/ftalat`.

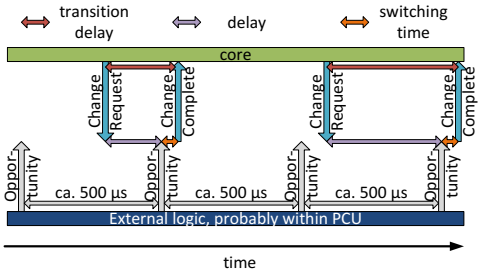


Figure 4.9: Per-core P-state mechanism in Haswell EP processors, based on [HSI+15]

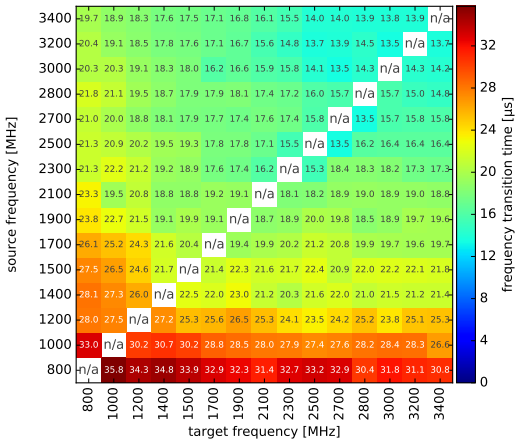
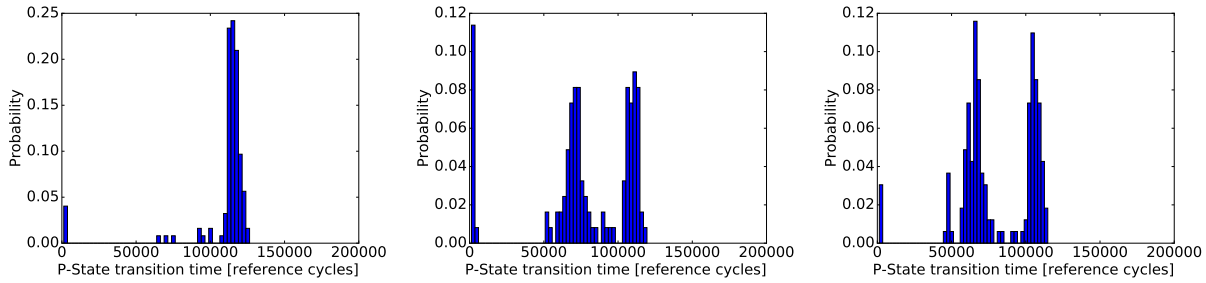


Figure 4.10: Frequency Transition delays for Haswell based desktop processor Intel Core i7-4770 (80 percent percentile)



(a) switch from 0.8 to 1.4 GHz (b) switch from 0.8 to 1.9 GHz (c) switch from 0.8 to 3.2 GHz

Figure 4.11: Frequency transition delay histograms measuring using FTaLaT on Haswell desktop processor Intel Core i7-4770

the server processor that has been described in the previous paragraph. As I show in Figure 4.11, some frequency transitions have two different clusters and additionally some results that are close to an instantaneous frequency switch. This behavior cannot be deducted from the information given in the respective processor manuals [Int14a, Int14b, Int13f]. When selecting the 80 percent percentile, as depicted in Figure 4.10, it is probable that the given sample represents the cluster with the highest transition latency. The results for this cluster are significantly different to those from previous desktop architectures or Haswell server processors. Usually, the frequency change latency of Desktop processors increases with a higher difference between source and target frequency when the P-state is decreased. For the examined Haswell desktop processor, the latency is highest when source and target frequency are low. Additionally, the worst case latency (35.8 µs) is very low compared to other desktop processors. A final distinction is that the test system provides similar latencies for increasing and decreasing the P-state. On previous architectures, lowering the frequency was less expensive than increasing it.

Intel Skylake

With the Intel Skylake desktop processor, the latency patterns of older processor generations are restored, as visualized in Figure 4.12. Again, lowering the frequency is significantly faster than increasing it. The highest latencies can be expected when increasing frequencies with a high difference between source and target frequency. However, there are some interesting patterns for this processor generation. If the frequency is to be increased and the source frequency is less than or equal to 1.7 GHz and the target frequency is greater than or equal to 2.2 GHz, the transition costs are almost constant for each given

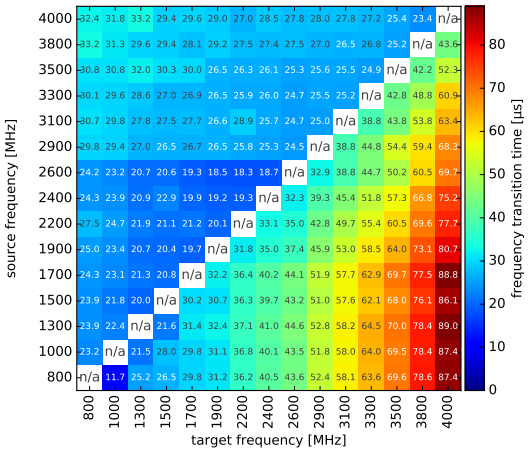


Figure 4.12: Frequency Transition delays for Skylake based desktop processor Intel Core i7-6700K

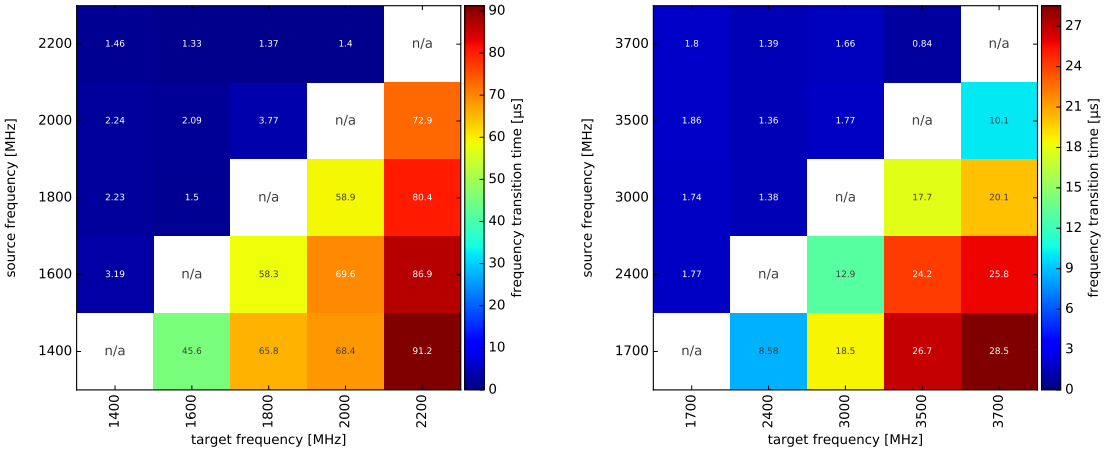
target frequency. I.e., a switch from 800 MHz to 4 GHz has the same latency as a switch from 1.7 GHz to 4 GHz. Additionally, frequency decreases with a source frequency of 2.6 GHz or less have a lower latency then those with a higher source frequency.

AMD Family 15h

AMD family 15h processors support two different voltage domains for computing resources: one for the computing modules (see Section 4.3.1) and one for the north bridge components. However, multiple computing modules can have different frequencies. Thus, the effect of switching frequencies and switching voltages is only loosely coupled and transparent to the user. The technical document [Adv13] describes in section 2.5.2.1.7 how a transition is executed:

> “ *If the P-state number is increasing (the compute unit is moving to a lower-performance state), then the COF [current operating frequency] is changed first, followed by the VID [voltage identifier] change. If the P-state number is decreasing, then the VID is changed first followed by the COF.* ”

Thus, a frequency decrease can be executed significantly faster than an elevation as the processor does not have to wait for a voltage change to complete. The results depicted in Figure 4.13 support this assumption. On both AMD processors, decreasing the frequency is one order of magnitude faster than on all examined Intel processors. However, these latencies cannot be used in the model as they are.



(a) Bulldozer based server processor AMD Opteron 6274

(b) Steamroller based desktop processor AMD A10-7850K

Figure 4.13: Frequency transition delays for AMD Family 15h based processors

The used algorithm is only able to measure frequency changes. Voltage changes are transparent to it. However, processor power consumption depends to a higher extend on voltage and less on frequency and thus the power consumption will change significantly when the VRs apply the new voltage. An upper limit for the expected voltage changes can be derived from the frequency increase results. Additional information can be gathered from the processor manuals [Adv13, Adv15, Adv16, Section 2.5.1.4]. In CSR D18F3xD8, the BIOS can program the Voltage Ramp Time, which is *"The maximum time to change VDD or VDDNB"* by a specific voltage difference.

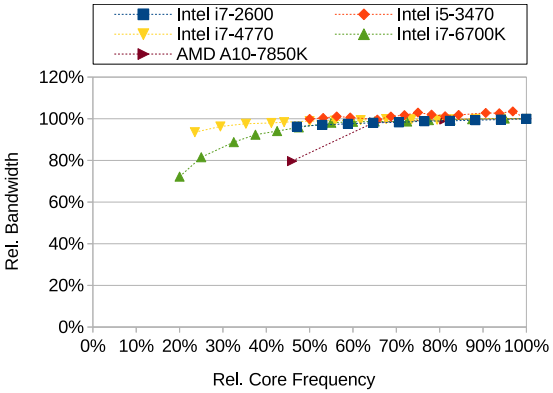
4.3.4 Performance Impact on Memory Bound Workloads

In [SHM12], I showed that processors with a separate uncore frequency can still provide a high main memory and L3-bandwidth if the core frequency is reduced. This is in accordance with the roof line model [WWP09]. In this section, I present bandwidth results for executing STREAM [McC95], which represents the most common benchmark for measuring the memory bandwidth of shared memory systems. The principal measurement algorithm is described in Algorithm C.1. The STREAM benchmark executes four different memory access patterns: Triad, Add, Copy, and Scale. The former two access patterns have a 1:1 ratio for read and write memory accesses. The latter two have a 2:1 ratio. I configure the parameters `NTIMES` and `STREAM_ARRAY_SIZE` in the following way: A shell script determines the Level-3 Cache size by reading the respective sysfs files. Afterwards, the script compiles the benchmark to use ten times the L3 cache size as memory size (`STREAM_ARRAY_SIZE`=$\frac{10*\text{L3 Cache Size}}{3*sizeof(double)}$). Then, it executes the benchmark on all cores of the first processor at the highest possible frequency. Based on the runtime of the benchmark, `NTIMES` is calculated so that the execution of the benchmark would finish within ten seconds. Then, the benchmark is executed for all available frequencies. To avoid increased memory latencies due to the cache coherence protocol in multi-processor systems, I prevent the second processor from going to a package idle state. Based on the performance impact, each STREAM execution will run for at least 10 seconds. From the resulting measurement times for the individual access patterns, I use the minimal times to report the bandwidth in dependence of the processor core frequency. I scale the frequency and the bandwidth to the reference frequency and the achieved bandwidth at reference frequency, respectively. Results for the access patterns Scale and Triad are illustrated in Figure 4.14.

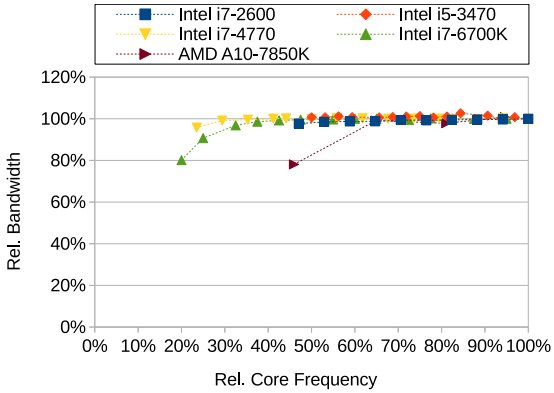
The measured main memory bandwidth of desktop processors is only negligibly influenced by the core frequency. Only three processors loose performance when the P-state is too high. The Haswell and Skylake processors Intel Core i7-4770 and Intel Core i7-6700K loose performance when the relative frequency is lower than 40 %, and 50 %, respectively. Both systems support a separate uncore frequency that is clocked independently by a hardware control loop [GSS15, Chapter 2]. When the uncore frequency is fixed manually at 4100 MHz, the performance loss vanishes, as depicted in Figure 4.15a. However, with a higher uncore frequency, the power consumption of the processor is significantly increased, as Figure 4.15b illustrates.

The AMD Family 15h processor AMD A10-7850K also looses bandwidth when the highest P-state is used even though the same uncore frequency is used for all P-states. Thus, it is probable that the different buffer sizes of the processor are not optimized for this core frequency.

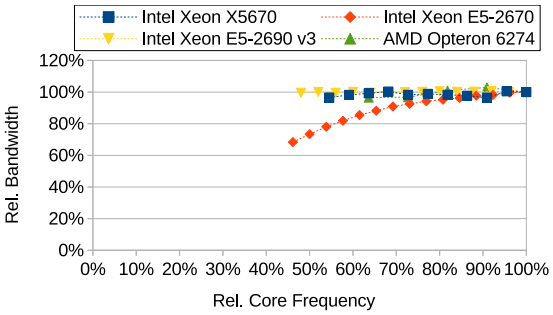
From the four investigated server processors, only two are unable to keep the memory performance if the frequency is reduced to a minimum: the Sandy Bridge-EP based Intel Xeon E5-2670 and the Family 15h based AMD Opteron 6274. On Intel Xeon X5670 and Intel Xeon E5-2670, access to data that is not available in the local L3 cache requires a request to the remote socket via QPI to check for updated data in its caches. This coherence mechanism of Sandy Bridge-EP processors likely scales with the common core/uncore frequency and therefore limits the memory request rate at reduced frequencies. In contrast, the fixed uncore frequency of the Westmere-EP based processor Intel Xeon X5670 enables high bandwidths even at reduced core frequencies. Likewise, the control loop of the Haswell-EP based processor Intel Xeon E5-2680 v3 is able to detect the memory accesses and can therefore maintain a high bandwidth. Still, a separate clock domain for the uncore is no guarantee for full memory bandwidth under reduced core frequencies as the results for AMD Opteron 6274 show. The complex uncore design with its



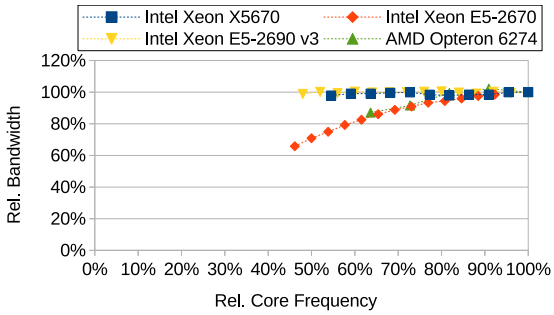
(a) Desktop processors – TRIAD



(b) Desktop processors – SCALE

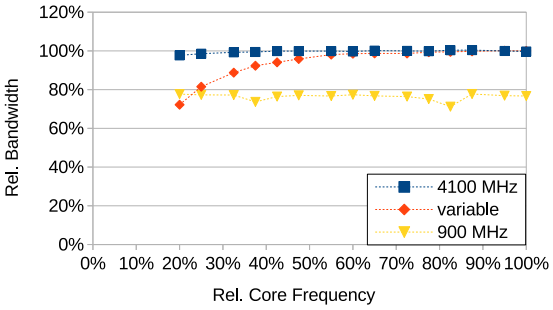


(c) Server processors – TRIAD

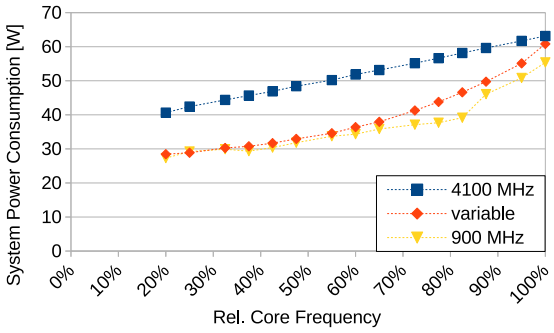


(d) Server processors – SCALE

Figure 4.14: Performance impact of frequency scaling on memory accesses for x86 processors. The figures illustrate the relative bandwidth scaled to the bandwidth at reference frequency of STREAM measurement kernels. Reducing the processor core frequency has only limited effect on the memory bandwidth.



(a) Relative bandwidth when executing TRIAD



(b) Average system power consumption when executing STREAM

Figure 4.15: Performance and power impact of different core and uncore frequency settings on STREAM benchmark for test system Intel Core i7-6700K.

various buffers and queues makes it hard to pinpoint the performance bottleneck. However, like for the desktop processor AMD A10-7850K, the memory bandwidth is reduced when the core frequency gets lower than 2.2 GHz. This has been measured with manipulating the respective MSRs that are described in Table 2.5 to re-define the existing P-states.

4.4 ACPI C-States

C-states on x86-processors can be entered with different mechanisms, e.g., by executing the *halt* instruction or using the instructions *monitor* and *mwait*. Such accesses are implemented in drivers within the Linux kernel, e.g., the intel_idle driver used by Intel processors or the acpi_idle driver for AMD processors. A common interface allows the cpuidle governor to use the predicted best state in a period of idleness. This interface can be used to instrument the various drivers at a higher level.

4.4.1 Scope

On all Intel systems, C-states are implemented per core. When Hyperthreading is active, the core C-state is defined by the lowest requested C-state of all hardware threads. The package C-state of Intel processors is defined by the lowest core C-state. On newer Intel processors (starting with the Haswell architecture), an additional package C-state (Package C2) is entered when the processor is in a deep package C-state but has to answer external requests (e.g., snoop traffic). This can increase the runtime of programs that are executed on other processors as they have to wait for the transition to PC2 until their request is answered [Int12b, Section 4.2.5]. On AMD, C-states are implemented per module. Thus, the minimal requested C-state from both cores is used.

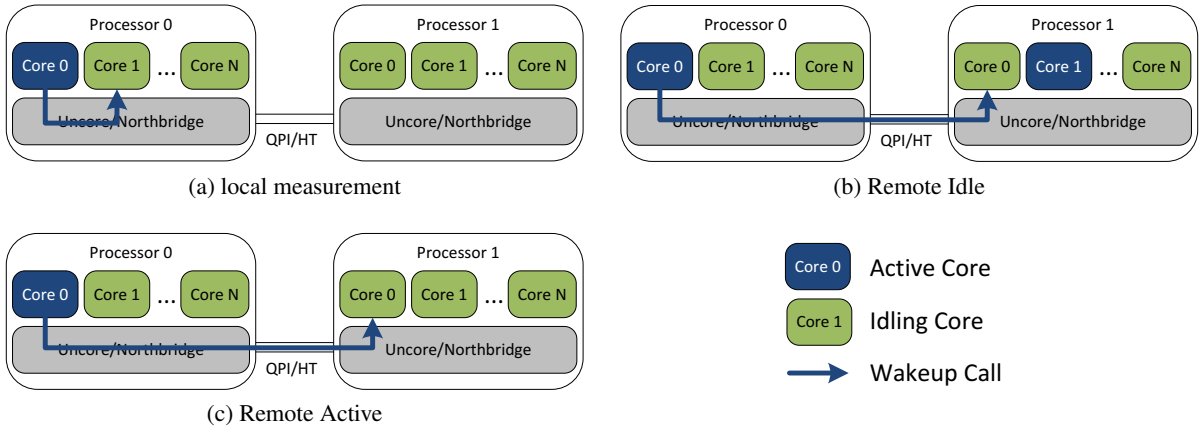
4.4.2 Access Costs $t(switch, c)$

Instructions that can be used to enter a C-state are usually only available in an operating system context and cannot be used from user space. Even though AMD enables vendors and administrators to enable a user-space access [Adv13, Section 5.13], this feature is barely used. Additionally, the processor halts after a C-state is initiated. This makes a measurement of the individual instructions impossible since there is no way of measuring the time after the instruction is completed.

4.4.3 Latency d

I presented the approach that is used in this section along the results for Westmere, Sandy Bridge and Bulldozer based server systems in [SMW14]. Results for the Haswell-EP system have been described in [HSI+15]. While it is possible to switch from every individual P-state to another one, C-state transitions are either from or to the active C-state CC0. The results in this section only target transitions *to* CC0. Measuring transitions to idle states would require to measure the time after a core or a processor enters an idle state. This cannot be done by the monitored core itself, since it would have to switch to *CC0* to measure the time. Alternatively, an external monitoring device could be used. However, the development of such a device is not part of this thesis.

To determine the latencies for the test systems, I patched the Linux kernel (3.13) such that it provides additional sysfs entries for each hardware thread. These entries are used to trigger the wake-up of the corresponding core, store the results of a single measurement, and constitute the interface between user- and kernel-space. Additionally, each hardware thread holds a data structure that is used for the measurement. In the following, I will refer to the processor core that initiates the wake-up sequence as *caller* and the processor core that is woken up as *callee*. For a measurement, a software thread that is pinned to the caller accesses the sysfs entry of the callee. Within kernel space, the software thread accesses the callee's data structure by setting a flag that the measurement should be taken and storing the current time. Afterwards, the thread calls the function *wake_up_nohz_cpu*, which triggers the waking of the callee. Every time a hardware thread returns from a C-state, it checks whether it should measure the current time. If its measurement flag is set, it stores the difference between the current time and the initiation time. Additionally, it stores the C-states it just left for filtering purposes. After the callee provided its data, the software thread that issued the wake-up reads the stored time difference and C-states.



(a) local measurement

(b) Remote Idle

(c) Remote Active

Figure 4.16: Measurement set-up for C-state transition latencies

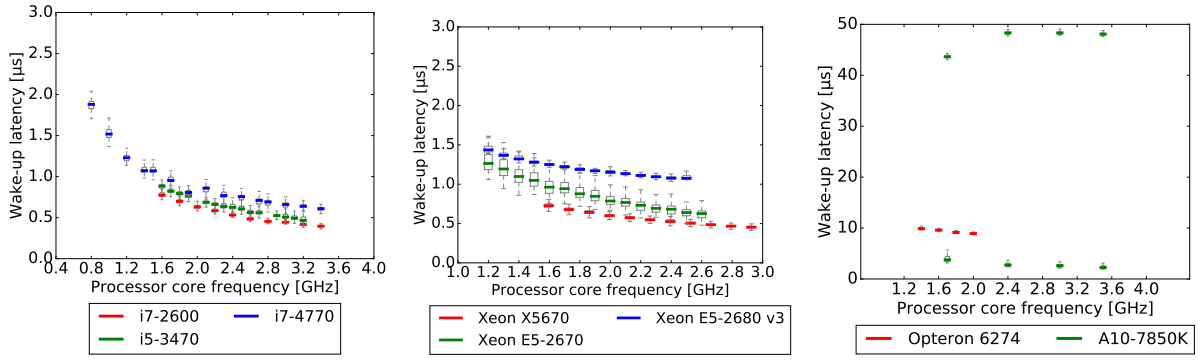
I use a Python script to take measurements for every C-state and frequency combination available on the system. The C-state of idle cores is controlled by disabling all higher C-states via sysfs entries[4]. Furthermore, the time between the single measurements is chosen high enough to enable the processor cores to use the highest available C-state.

The script collects the wake-up data for *Local* and, if the system hosts multiple processors, *Remote* measurements. In the *Local* case the caller and the callee are located on the same processor but on different cores. In the *Remote* case the callee is located on a different socket. Since the package C-state is not controllable via software, I make two distinct measurements on other packages called *Remote Idle* and *Remote Active*. The difference between these is that when measuring *Remote Active*, I enable a busy waiting loop on an additional core on the package of the callee. Thus, the package is unable to enter a package C-state. All different measurement possibilities for a two socket system are depicted in Figure 4.16.

I measure each combination of C-state and P-state at least 400 times and filter results where the operating system requested a C-state that differs from the highest possible one. To do so, I provide the selected C-state in another *sysfs* file. This occurred for less than one percent of the samples. The AMD test system AMD Opteron 6274 implements two dies per processor. As the two dies within one processor package share some resources, they might also influence each other's idle behavior. I therefore measure the remote wake-up latencies for a core on the second die in the package (*Near Remote*) as well as a core in another package (*Far Remote*).

In Figure 4.17, I show that the transition latency for CC1 states depends on the processor frequency that is applied. In the measurement setup, all processor cores share the same frequency. A lower frequency correlates with a higher transition latency on all test systems, except the AMD A10-7850K, which is discussed later. The latency is influenced by an increased message latency between the cores as well as the slower execution of the instructions that are executed during the wake-up procedure. Surprisingly, the latency increases with newer Intel architectures compared to older ones. This can be seen for desktop processors as well as for server processors. Desktop processors mostly have a lower latency than their server counter parts. While the expected range is between 0.4 µs and 0.8 µs on the Sandy Bridge based Intel Core i7-2600, the Intel Xeon E5-2670 test system, which is also based on the Sandy Bridge architecture, shows latencies between 0.6 and 1.4 µs. This can partially be attributed to the different frequency ranges. For the Haswell based systems Intel Core i7-4770 and Intel Xeon E5-2680 v3, the desktop system has a higher range of possible latencies. This can be attributed to the frequency range, since the latency increases significantly for lower frequencies that are not available on the server system. For equal frequencies, the desktop processor provides lower latencies. Compared to the Intel processors, the AMD test systems have a significantly higher C1-latency. Additionally, the results of the desktop

[4] `/sys/devices/system/cpu/cpu*/cpuidle/state*/disable`



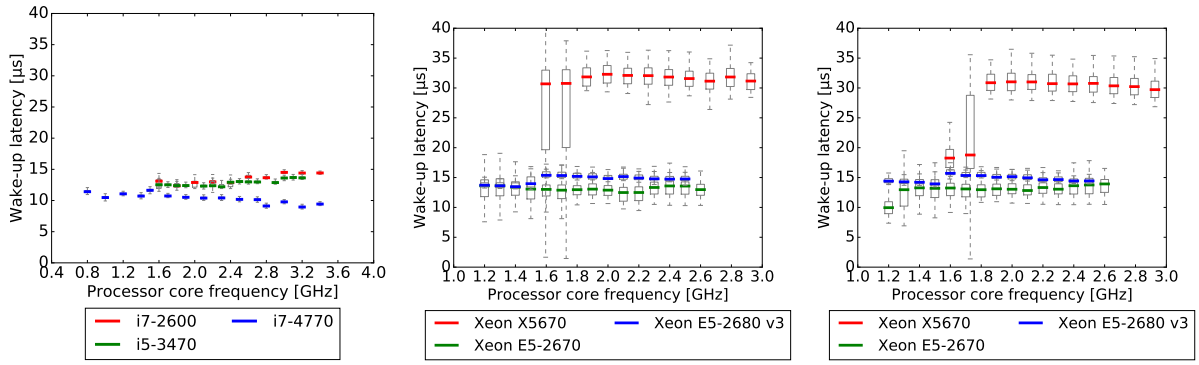
(a) Intel desktop processors (b) Intel server processors (c) AMD processors

Figure 4.17: CC1 (halt) state for different processors (Y-Axis scale depending on processor vendor.)

processor AMD A10-7850K have two clusters. Approximately half of the results are within the range of 2 to 4 µs. Within this cluster, the latency decreases with the applied processor core frequency. The other half of the obtained results has a latency of 43 to 49 µs, where the lowest latency is achieved by the lowest core frequency. An explanation for this is given in connection with the CC6 results discussed later on.

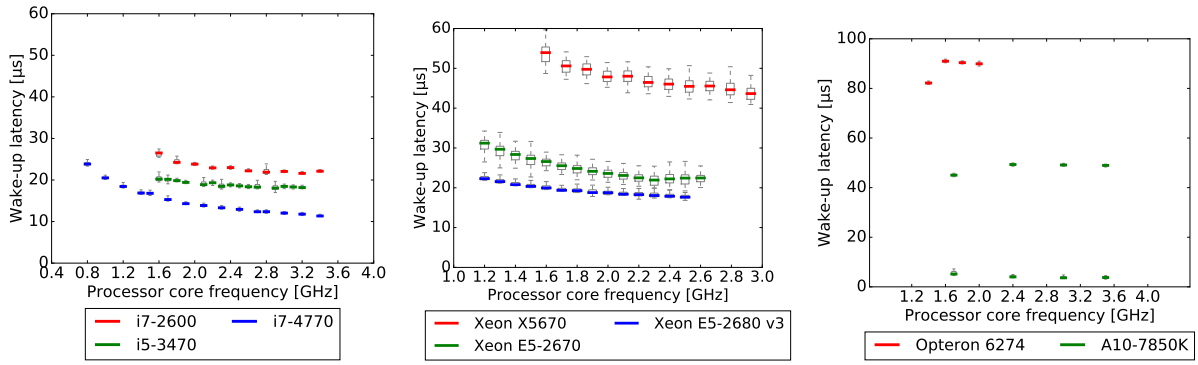
Latency results for the Intel CC3 state are depicted in Figure 4.18. In contrast to the CC1 latencies, CC3 latencies are not influenced by the processor core frequency. The architectural switch from Westmere (Intel Xeon X5670) to Sandy Bridge (Intel Xeon E5-2670) reduced the CC3 latency significantly from approx. 32 to 12 µs. Later architecture changes have almost no influences on the performance. While the Intel Core i7-4770 test system provides slightly better results compared to its Sandy Bridge and Ivy Bridge based counterparts, the Haswell server processor Intel Xeon E5-2680 v3 has a higher latency compared to Intel Xeon E5-2670.

CC6 transition latencies for the test systems are given in Figure 4.19. Again a significant improvement can be seen when comparing the Westmere based Intel Xeon X5670 to the Sandy Bridge based Intel Xeon E5-2670. The Haswell architecture reduces the CC6 latency additionally. For desktop processors newer architectures also correlate with a lower CC6 latency. The results for CC6-latencies of AMD processors are depicted in Figure 4.19c. Like for the CC1 results (Figure 4.17c), the desktop system has two different result clusters for each frequency. For one cluster, the latencies decrease with an increased processor core frequency. The core returns to CC0 after 7 µs at 1.7 GHz and 3.5µs at 3.6 GHz. In the other cluster, the lowest core frequency provides the lowest CC6-latency. This behavior can also be seen for CC6 states on the server processor AMD Opteron 6274. The different clusters result from a hardware



(a) Intel desktop processors (b) Intel server processors (Local) (c) Intel server processors (Remote)

Figure 4.18: CC3 state for Intel processors

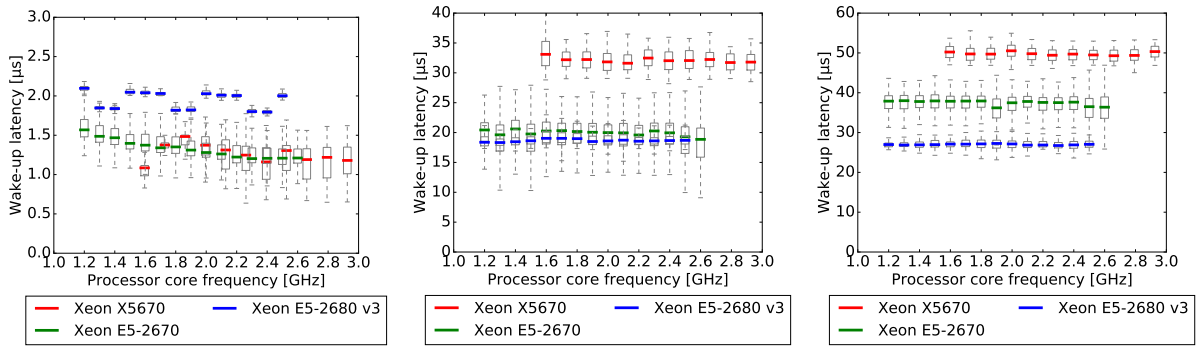


(a) Intel desktop processors (b) Intel server processors (c) AMD processors

Figure 4.19: CC6 state for different processors (Y-Axis scale depending on processor vendor)

internal mechanism that decides at runtime which C-state should be applied. According to the C-state control register (see Table 2.4) the only difference between the two states is the definition of the flush timer, which triggers after 143 and 512 µs for CC1 and CC6, respectively. Thus, either an assumed CC1 state (with low latencies) or a CC6 state (with high latencies) is used, independent from the requested C-state. For these effective C-states, the latencies of the desktop system are approximately half of the ones that are observed for the server processor.

Transition latencies from Package C-states are depicted in Figure 4.20. Returning from C1E is slightly slower than from C1 but still faster than 2.5 µs. The Haswell based Intel Xeon E5-2680 v3 has a higher latency than the other processors. The difference between the Sandy Bridge based Intel Xeon E5-2670 and the Westmere based Intel Xeon X5670 is within the measurement variation. The performance of the Intel Xeon E5-2680 v3 processor shows two different latency levels that depend on the frequency. This cannot be related to measurement errors with arbitrary medians as the deviations are too small. Thus, this effect is related to the core frequency but its source cannot be explained. The PC3 latency is significantly reduced from approx 32 µs to 19 µs from the Westmere processor generation to the Sandy Bridge generation. For the Westmere based system, results for the CC3 and the PC3 state are almost identical. However, the outliers for lower frequencies that could be seen in the core C-state are not present when the package C-state is measured. For the newer processors Intel Xeon E5-2670 and Intel Xeon E5-2680 v3, the latency increases by approx. 5 µs compared to CC3. As expected, Package C-state 6 causes the highest latency on the Intel systems. Compared to PC3, the transition time is increased by approx. 18 µs on the Westmere and Sandy Bridge based system, and by 10 µs on the Haswell based Intel Xeon E5-2680 v3.



(a) C1E state on Intel processors (b) PC3 state on Intel processors (c) PC6 state on Intel processors

Figure 4.20: Package C-states for Intel processors (Y-Axis scale fits the respective Core C-state)

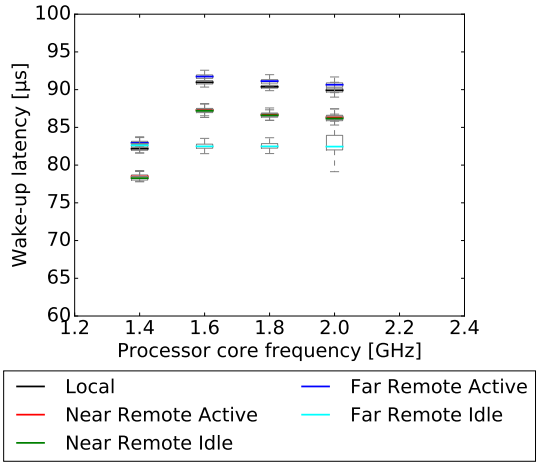


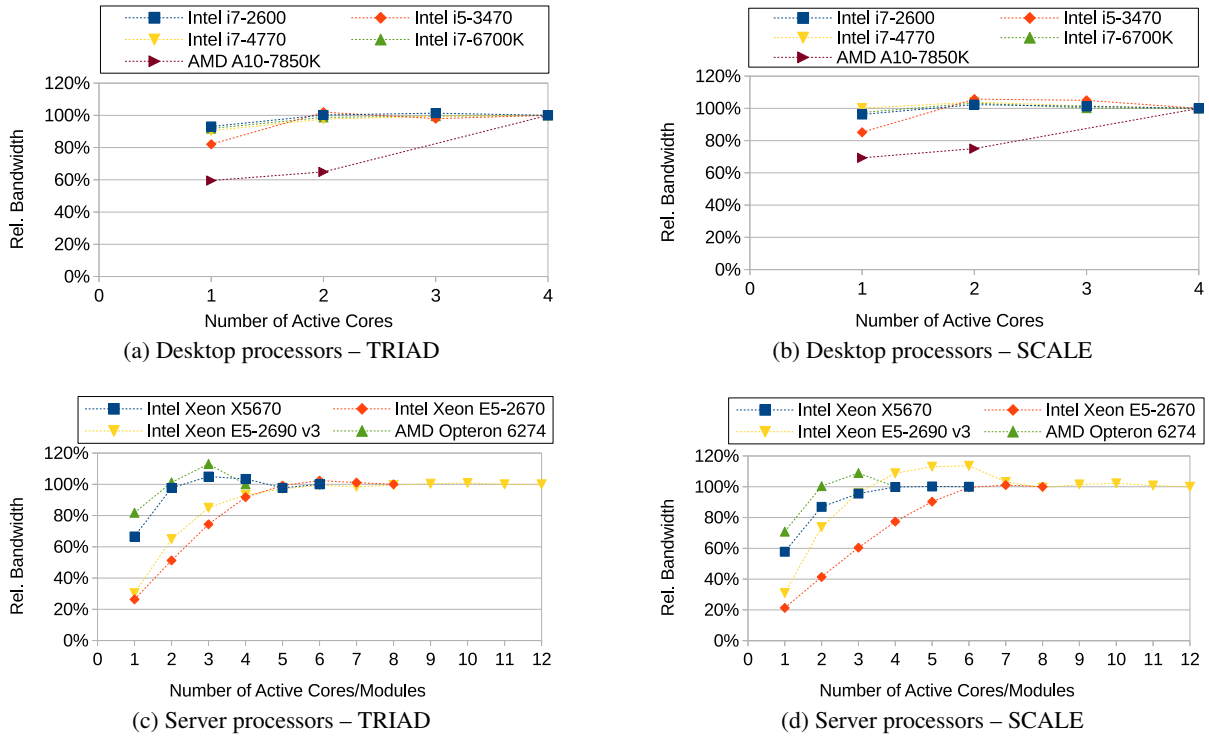
Figure 4.21: Different C6 latencies on AMD Opteron 6274 test system. The latency increase from 1.4 to 1.6 Ghz does not appear for *Far Remote Idle* accesses.

Figure 4.21 depicts latency results for the Package C6 state of the AMD Opteron 6274 test system. For most measurement scenarios, the lowest latency can be seen at the lowest applied core frequency. This could be caused by additional frequency transitions that are required at higher frequencies as the CC6 state reduces the frequency to the one defined by the highest P-state[5][6]. The wake-up latency for *Near Remote* cases is faster than the *Local* one. I attribute that to the distance between the callee and the processor that holds the C-state information in its DRAM. In this case, the local processor is two HyperTransport hops away from this package while the near-remote processor is only one hop away. As Molka et al. show in [MHS14], this distance has a high influence on the latency and bandwidth between different processors. Thus, it is faster for the near-remote processor to re-establish its processor state. The *Far Remote* processor is also multiple hops away.
Independent of the distance, the behavior for cores in other processors is identical if there is one active core in the callee's package. In that case, the core voltage shared by all compute units is already at the required level for the requested frequency. Thus, the frequency transition is initiated immediately and contributes to the measured latency[7]. The situation is different for completely idling processors, which is depicted as *Far Remote Idle*. In that case, all cores entered the CC6 state. Consequently, the voltage is reduced to the required voltage for the lowest frequency. Therefore, the frequency transition has to be delayed until the voltage has been ramped up to the required level. However, the processing of instructions continues during the voltage ramp using the lower frequency. Therefore, the callee completes its operations before the target frequency is restored. Anyway, the frequency change is only delayed. The associated overhead is just not included in the measurement.

4.4.4 Performance Impact on Memory Bound Workloads

Treibig and Hager describe limiting factors when accessing memory in [TH10]. They show that the achieved bandwidth does not necessarily scale with the number of accessing processor cores. Thus, when applying concurrency throttling [CMBAN08] (i.e., reducing the number of active cores), the energy efficiency of a parallel program can be increased in memory bandwidth bound regions. In [SHM12], I showed how a varying concurrency influences the read bandwidth for main memory and L3-cache streaming accesses. To achieve a lower power consumption, the unused cores can use idle states. In this section, I describe the effect of such a C-state optimization on the achieved bandwidth. To do so, I use the same methodology as described in Section 4.3.4, but instead of changing the frequency, I reduce the number of concurrent threads. Figure 4.22 depicts the effect of reduced concurrency on the bandwidth for the test systems that are described in Appendix A.

[5] D18F3 xA8[31:29] PopDownPstate = D18F3 xDC[10:8] HwPstateMaxVal
[6] each module has its own frequency domain while all modules share one voltage domain
[7] the processing of instructions is stopped during the frequency transition as there is no stable clock signal



(a) Desktop processors – TRIAD

(b) Desktop processors – SCALE

(c) Server processors – TRIAD

(d) Server processors – SCALE

Figure 4.22: Performance impact of concurrency scaling on memory accesses for x86 processors. The figures illustrate the relative bandwidth scaled to the bandwidth at full concurrency of STREAM measurement kernels. For most architectures, the number of active cores can be reduced with a limited effect on the memory bandwidth.

Figure 4.22a and Figure 4.22b show the relative memory bandwidth of desktop processors for different concurrencies at reference frequency. While Intel processors reach at least 80 % of the bandwidth with only one active core, the AMD A10-7850K test system achieves only 60 % for TRIAD and 70 % for SCALE. Even if both cores of one module are used, the performance does not increase significantly. Thus, both modules have to be active to achieve the full bandwidth.
The performance of server processors is depicted in Figure 4.22c and Figure 4.22d. Here, the concurrency can be decreased significantly without reducing the overall bandwidth. However, individual test systems have different points for the optimal concurrency setting. On the Intel Xeon E5-2670 test system, at least five cores have to be used to achieve the full bandwidth for the TRIAD measurement kernel. The number of cores has to be increased to at least six for the SCALE operation. This can be attributed to different weights of load and store operations. On other systems, a larger share of cores or modules can be de-activated without hurting the overall performance: for the TRIAD operations, only two out of four modules of the AMD Opteron 6274 test system, two out of six cores on Intel Xeon X5670, and five out of twelve on Intel Xeon E5-2680 v3. In some cases, the bandwidth can be increased by reducing the concurrency. In [AMD12], AMD states that a lower number of threads can indeed increase the achievable bandwidth. This is confirmed with the depicted measurements. The Intel Xeon E5-2680 v3 test system shows a similar behavior for the SCALE operation.

4.5 ACPI T-States

According to processor manuals, ACPI T-states use clock modulation to lower the power consumption of processors [Int15a, Chapter 14.7.2]. The initial intent for implementing such a clock modulation was to prevent processors from overheating. The first implementation *Thermal Monitor 1* (TM1) has been introduced with Intel Pentium 4 processors. Another feature that enforces a low processor temperature is called *Thermal Monitor 2* (TM2). It has been introduced with Intel Pentium M processors and uses DVFS to achieve a lower power dissipation. In [RNMM04], Rotem et al. describe how this technique is more effective than its clock modulation counterpart. Starting with Intel Core 2 processors, Intel introduced *Adaptive Thermal Monitor* where different transition targets can be used under overheating conditions. Processor manuals [Int15a] describe this mechanism to dynamically select between TM2 and TM1. According to technical documents for desktop processors [Int09, Int13a], it applies DVFS when the target temperature is exceeded and clock modulation in addition if the thermal effect of DVFS is not sufficient. Clock modulation can also be triggered by software via the MSR register `IA32_CLOCK_MODULATION`. This interface can be used to modify the percentage of skipped cycles in steps of 6.25 % (12.5 % for older architectures). This percentage is also called the duty cycle of the clock modulation assertion signal [BP06, Section 4.1.3], which is defined as the quotient of the pulse duration and the period length. Both are depicted in Figure 4.23. In the following, I will use the notation described in [SIB+16]. The used parameters are: f – the current frequency of the system, m – the current duty cycle setting, $\Delta t_{thr}(f, m)$ – the pulse duration of the clock modulation signal, and $T_{thr}(f, m)$ – the period length of the clock modulation signal.

4.5.1 Scope

Like core C-states, T-states are implemented per processor core. However, when hardware threads do not synchronize their settings, the processor core uses the last setting that was written to the register to determine the current T-states.

4.5.2 Low Level Implementation Details

To interpret the results in the following sections, low level information about the clock modulation implementation is necessary. In this section, I describe how I measured pulse duration and the period length of the test systems and how the different processors performed. I presented these findings to the scientific community in [SIB+16].
To determine how clock modulation is implemented by the processor, I use a measurement routine that is similar to the one used by Mazouz et al. for measuring P-state latencies [MLPJ14], which has also been used in Section 4.3.3.

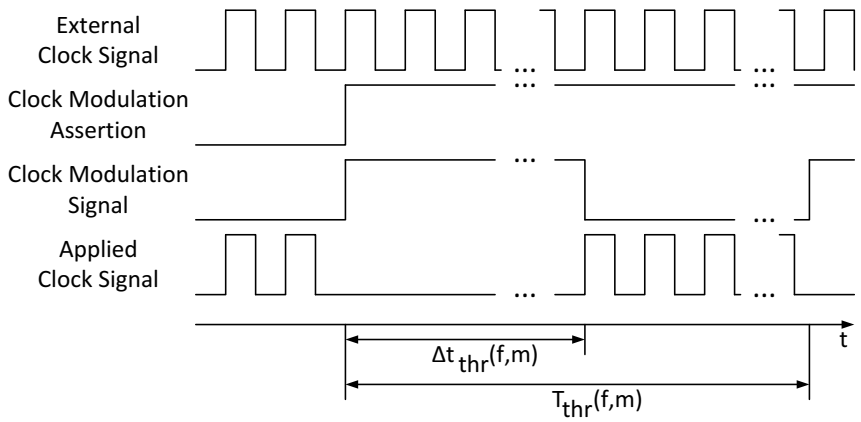


Figure 4.23: Parameters of clock modulation signal

```
unsigned hi, lo; unsigned long long count;
for ( count = 0 ; count < STORE_SIZE ; count++ )
{
        asm volatile( // 150 adds
                "addl $1,%%eax;"
                "addl $1,%%eax;"
                //... repeat more adds
                :::"%eax");
        // get time in reference cycles
        asm volatile("mfence;rdtsc":"=a"(lo),"=d"(hi));
        // store results
        results[count] = ((unsigned long long) lo) |
                (((unsigned long long) hi) << 32);
}
```

Listing 4.1: Measurement loop for short timescale analysis

The used implementation is shown in Listing 4.1. I run 2^{20} iterations of this measurement loop and record the respective run times in memory. The first 75 % of the results are skipped in the analysis, as they exhibit more noise than the latter 25 %. This results in a total number of 2^{18} samples. To control the clock modulation setting, I use the x86_adapt library and kernel module, which I describe in Section 6.1.
The entire benchmark is repeated with all combinations of available frequencies and clock modulation settings. Additionally, the clock modulation is set either on one or on all cores of a system to check whether the implementation treats this differently. The gathered run-times are analyzed post-mortem.
Based on the measurements, one can determine the following parameters: $t_{std}(f,m)$ represents the time spent for executing one benchmark loop. $t_{thr}(f,m)$ represents the execution time when the loop is interrupted by a throttling event. The pulse duration $\Delta t_{thr}(f,m)$ can be described as the difference between $t_{thr}(f,m)$ and $t_{std}(f)$. $T_{thr}(f,m)$ represents the time between two throttling events and the period length of the clock modulation signal. Two throttling cycles with parameters and measured times are depicted in Figure 4.24.
From the captured runtimes $t^i(f,m)$, I first determine $t_{std}(f,m)$. To do so, the runtimes are sorted in ascending order and a cluster is created, starting with the minimum runtime $t^0(f,m)$. The next runtime $t^{i+1}(f,m)$ is added to this cluster as long as $t^{i+1}(f,m) - t^i(f,m) < \max(d_{abs}, t^i(f,m) \cdot (1 + d_{rel}))$, where $d_{abs} = 30$ ref. cycles and $d_{rel} = 8\,\%$. The minimum and maximum of this cluster is denoted as $t^{min}_{std}(f,m)$ and $t^{max}_{std}(f,m)$, respectively. Afterwards, another cluster for $t_{thr}(f,m)$ is searched. The same algorithm as before is used and the minimal time $t^{min}_{thr}(f,m)$ is set to the successor of $t^{max}_{std}(f,m)$. If the resulting cluster does not represent at least 5 % of the overall runtime, $t^{min}_{thr}(f,m)$ is increased and the search is repeated until the 5 %-criterion is met. Runtimes that are not part of these clusters are classified as outliers. To further remove outliers within the clusters, their medians are used as $t_{std}(f,m)$ and $t_{thr}(f,m)$. To determine the distance between two throttling events $T_{thr}(f,m)$, I go through the initial unsorted list of measured runtimes and identify the time difference between loops with a runtime $t^i(f,m)$, where $t^i(f,m) >= t^{min}_{thr}(f,m)$ and $t^i(f,m) <= t^{max}_{thr}(f,m)$. Afterwards, the median of these time differences is defined as $T_{thr}(f,m)$.

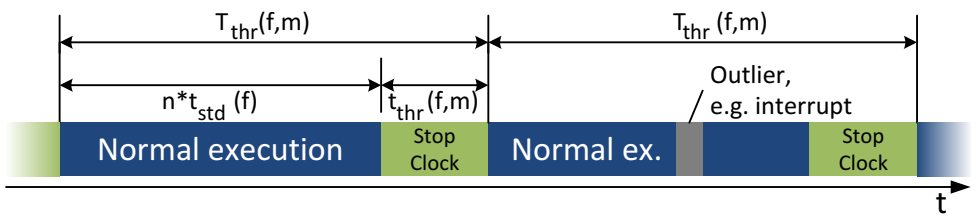
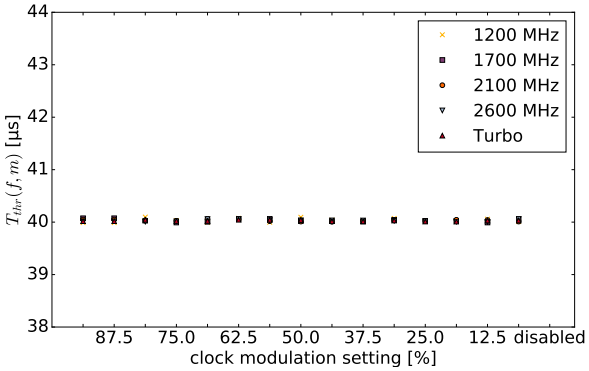
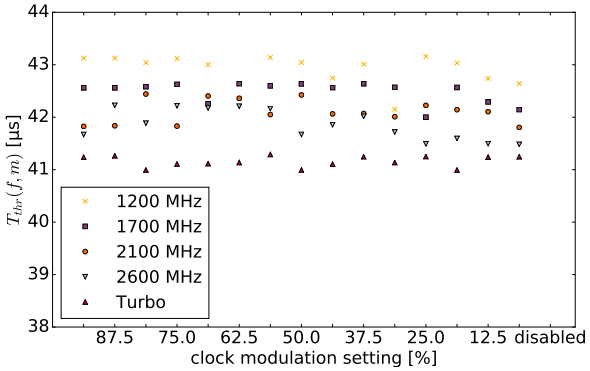


Figure 4.24: Clock modulation, measured parameters



(a) On Sandy Bridge and Ivy Bridge test systems, $T_{thr}(f, m)$ is not influenced by f and m. The figure shows results from the Sandy Bridge EP sytem.

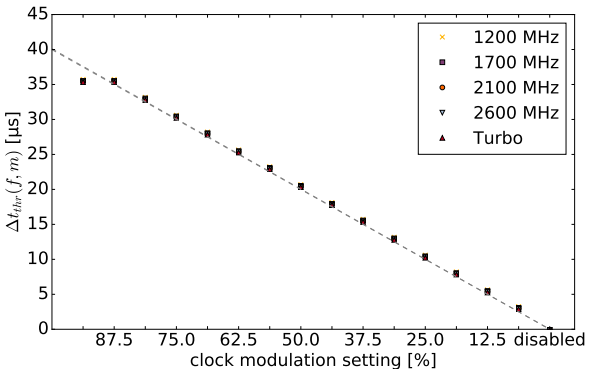


(b) On Haswell and Skylake processors, $T_{thr}(f, m)$ varies significantly and is influenced by the applied frequency f. The figure depicts results from a Haswell-EP test system.

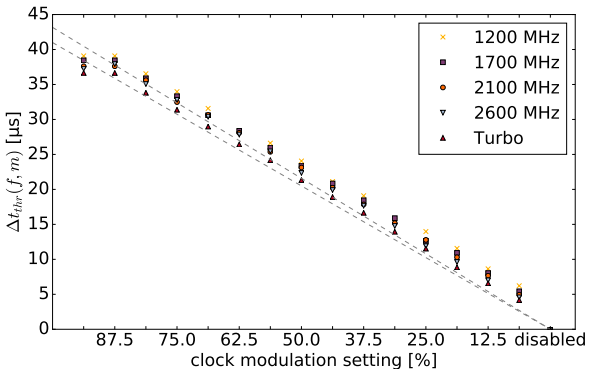
Figure 4.25: The clock modulation window $T_{thr}(f, m)$, which includes a period of clock stop assertion and clock stop desertion, is between 40 and 45 microseconds on all examined architectures.

The time between two throttling events $T_{thr}(f, m)$ is independent from f and m on Sandy Bridge and Ivy Bridge processors and varies significantly between 40.5 and 43.5 µs on Haswell and Skylake processors. This is depicted in Figure 4.25. On the newer architecures, $T_{thr}(f, m)$ increases with a lower core frequency.

In an ideal implementation, the clock modulation setting m will be directly translated to the pulse duration of the clock modulation signal $\Delta t_{thr}(f, m)$ so that the processor will not execute cycles for the respective share of $T_{thr}(f, m)$. For a theoretical clock modulation setting of 100 %, this share would result in $\Delta t_{thr}(f, m)$ being $T_{thr}(f, m)$ while for disabled clock modulation $\Delta t_{thr}(f, m)$ would be 0. The remaining clock modulation settings should provide a $\Delta t_{thr}(f, m) = m * T_{thr}(f, m)$. Figure 4.26 shows that the resulting throttling times do not follow the ideal, but $\Delta t_{thr}(f, m)$ is higher than expected for all clock modulation settings < 93.75 %. On Haswell and Skylake architectures, the difference between the ideal and measured throttling time decreases with a higher clock modulation setting while it is almost constant for Sandy Bridge and Ivy Bridge processors. Furthermore, the highest clock modulation setting



(a) On Sandy Bridge processors, $T_{thr}(f, m)$ is constant for all f and m. Thus, the lines for the minimal and maximal expected $\Delta t_{thr}(f, m)$ overlap. $\Delta t_{thr}(f, m)$ is higher than expected, except for m=93.75 %.



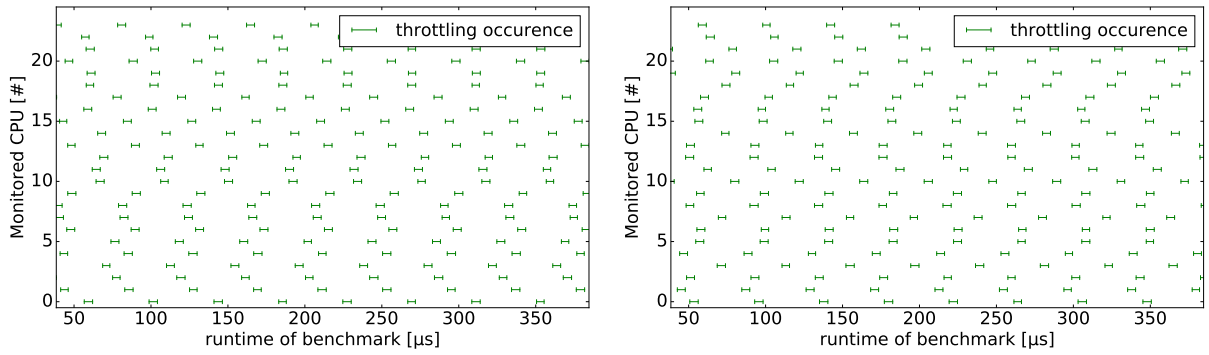
(b) $T_{thr}(f, m)$ varies significantly on Haswell processors, depending on f and m. Still, most of the measured $\Delta t_{thr}(f, m)$ are above the expected range.

Figure 4.26: The clock modulation signal assertion time $\Delta t_{thr}(f, m)$ is higher than described in the processor manual with the exception of a 93.5 % clock modulation setting. The gray dashed lines depict the expected maximal and minimal $\Delta t_{thr}(f, m)$, based on the assumption that $\Delta t_{thr}(f, 100\%) = T_{thr}(f, m)$ and $\Delta t_{thr}(f, disabled) = 0$.

Table 4.3: Intel Xeon E5-2670 loop runtimes. When all cores apply a common clock modulation setting, DVFS is used alternatively which increases t_{std}. This behavior applies only to Sandy Bridge and Ivy Bridge processors.

Freq. [MHz]	cores	Result	(93.75)	ex. 2				**Clock modulation setting [%]**						ex. 1		dis-
			87.5	81.25	75	68.75	62.5	56.25	50	43.75	37.5	31.25	25	18.75	12.5	6.25	abled
2600	one	t_{std}[µs]	0.0831	0.0831	0.0831	0.0831	0.0831	0.0831	0.0831	0.0831	0.0831	0.0831	0.0831	0.0831	0.0831	0.0831	0.0831
	all		0.18	0.18	0.18	0.18	0.18	0.18	0.166	0.153	0.135	0.127	0.112	0.105	0.0969	0.09	0.0831
	one	t_{thr}[µs]	35.5	32.9	30.4	28	25.4	23	20.4	17.9	15.5	12.9	10.4	7.97	5.41	3.01	-
	all		28.4	23.4	18.3	13.3	8.38	-	-	-	-	-	-	-	-	-	-
2000	one	t_{std}[µs]	0.108	0.108	0.108	0.108	0.108	0.108	0.108	0.108	0.108	0.108	0.108	0.108	0.108	0.108	0.108
	all		0.18	0.18	0.18	0.18	0.177	0.18	0.18	0.18	0.18	0.166	0.144	0.135	0.127	0.12	0.108
	one	t_{thr}[µs]	335.6	33	30.5	28.1	25.5	23.1	20.6	18	15.6	13	10.5	8.07	5.51	3.11	-
	all		30.8	28.4	23.4	18.3	15.9	10.8	5.82	3.42	-	-	-	-	-	-	-
1200	one	t_{std}[µs]	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.177	0.18	0.18	0.18	0.18	0.18	0.18
	all		0.18	0.18	0.18	0.177	0.18	0.18	0.18	0.18	0.177	0.18	0.18	0.18	0.18	0.18	0.18
	one	t_{thr}[µs]	35.9	33.3	30.8	28.4	25.8	23.4	20.9	18.3	15.9	13.3	10.8	8.38	5.82	3.42	-
	all		35.9	33.3	30.8	28.4	25.8	23.4	20.9	18.3	15.9	13.3	10.8	8.38	5.82	3.42	-

(93.75 %) provides the same results as the second highest (87.5 %). This can be seen for all architectures except the Skylake desktop processor, where the final clock modulation step increases $\Delta t_{thr}(f, m)$ by approx. 2 % (depending on the frequency) compared to the previous setting of 87.5 % (not depicted).
These results can also explain the significant energy efficiency gains when using clock modulation on Sandy Bridge or Ivy Bridge processors as presented in [BPP15, CTC14], and [WSM15]. If the clock modulation setting is equal among *all cores* and can be represented as a frequency (i.e., if the original frequency is high enough and the clock modulation is not too high), the processor uses DVFS instead. For example, for f=2600 MHz, the runtime t_{std} of the loop is 83.1 ns, which does not change when a single core applies clock modulation. This can be seen in Table 4.3. When all cores set their clock modulation value to a common value, e.g., 12.5 %, t_{std} increases to 96.9 ns, which would correspond to f=2230 MHz. No clock modulation can be observed. When the targeted performance of the common clock modulation setting is lower than the lowest supported frequency, clock modulation is used in addition to DVFS. For example, if a clock modulation setting of 81.25 % is applied to a frequency of 2600 MHz, the standard runtime t_{std} increases to 180 ns, which indicates a processor frequency of 1200 MHz. Here, DVFS reduces the performance to 46.2 %. In addition, a clock modulation time t_{thr} of 23.4 µs is introduced, which reduces the total average performance to 19.2 % relative to the baseline with no clock modulation. This is close to the 18.75 % performance target. Please note that this behavior only applies to Sandy Bridge and Ivy Bridge processors and could not be observed on any other test system.
The final observation of the low level analysis targets the synchronicity of throttling events. Here, I use an OpenMP parallel version of our measurement loop and store the runtimes of each thread. Figure 4.27 depicts measurements of the Haswell-EP system with f=2000 MHz and m=6.25 %. The results indicate that there is no synchronization between the clock modulation mechanisms of the single cores. Furthermore, repeated experiments result in a completely different pattern.



(a) Measurement 1 (b) Measurement 2

Figure 4.27: Clock modulation pattern for all cores on a dual socket Haswell EP system.

4.5.3 Access Costs $t(switch, c)$

The cost of accessing T-states depends on the time when the switch is started relative to the clock modulation signal. If the access is not interrupted by a clock gating, (i.e., if the clock modulation signal does not become active), the access time is equal to the access time with no clock modulation. Otherwise, it is increased by $\Delta t_{thr}(f, m)$ for every clock modulation assertion. The average execution time of a switch can be calculated with the following equation:

$$t(switch, m) = \overbrace{\frac{t(switch, m = 0.0\,\%)}{T_{thr}(f, m) - \Delta t_{thr}(f, m)}}^{\text{number of clock modulation assertions}} * \Delta t_{thr}(f, m) + t(switch, m = 0.0\,\%) \tag{4.1}$$

However, the total number of interrupting clock modulations is the integer part of $t(switch, m)$ in Equation 4.1 plus an additional phase that has a probability of the decimal part of $t(switch, m)$.
To measure the average access times for T-state changes, I use the same algorithm as described in Section 4.3.2. The results are listed in Table 4.4.

4.5.4 Latency d

In this section, I distinguish between latencies when enabling T-states (i.e, setting the duty cycle to m) and disabling T-states (i.e., setting the duty cycle from m to 0 %). To gather the initialization delay, I (1) get the expected runtime t_{std} of the measurement loop, (2) activate clock modulation, and (3) execute the loop until the measured runtime is significantly higher. Afterwards,I ensure that the extended runtime $t^i(f, m)$ is within the expected range, i.e., $t^{min}_{thr}(f, m) <= t^i(f, m) <= t^{max}_{thr}(f, m)$. To measure the delay after deactivating clock modulation, I (1) wait a random time after the last clock modulation cycle, (2) deactivate clock modulation, and (3) wait for up to 60 µs for an extended runtime. I register the random wait-time, the extended runtime and the time between deactivating clock modulation and the start time of the interrupted loop.
On the Sandy Bridge and Ivy Bridge architectures, the first clock modulation is executed 12.5 µs after its activation by software. On newer architectures, this initialization delay is $T_{thr}(f, m) - \Delta t_{thr}(f, m)$. Here, the activation trigger can be seen as falling edge of the clock modulation signal. On the Sandy Bridge-EP processors, the assertion is deactivated 17 µs after software triggers the register. Thus, clock modulation phase can be executed (partially) after it has been disabled. The same behavior can also be observed on the other examined Sandy Bridge and Ivy Bridge processors. On Haswell and Skylake processors, no clock modulation activity could be observed after T-states were disabled.

4.5.5 Performance Impact on Memory Bound Workloads

While the clock modulation signal is active, processor cores can be considered to be clock-gated, which is comparable to C-state mechanisms. Thus, if a number of cores resides in such a state, the memory performance is reduced according to the number of cores that are still active. However, I described in Section 4.5.2 that the clock modulation phases of the different cores are not synchronized. Thus, over a time frame, there is a binomial distribution of how many cores are active concurrently. This is depicted in Figure 4.28. The resulting expected bandwidth can now be calculated as the weighted average of the bandwidths for the different number of active cores. The weight for each of the single bandwidths corresponds to the probability of the number of cores being active for the given m.

	Duty cycle [%]															
Test system	disabled	6.25	12.5	18.75	25.0	31.25	37.5	43.75	50.0	56.25	62.5	68.75	75.0	81.25	87.5	93.75
Intel Xeon X5670	0.37	-	0.43	-	0.5	-	0.6	-	0.75	-	1.0	-	1.48	-	2.95	-
Intel Xeon E5-2680 v3	0.58	0.65	0.7	0.75	0.81	0.89	0.99	1.09	1.22	1.42	1.65	1.98	2.45	3.47	5.23	5.23

Table 4.4: Average runtime for changing T-states via MSR interface in µs

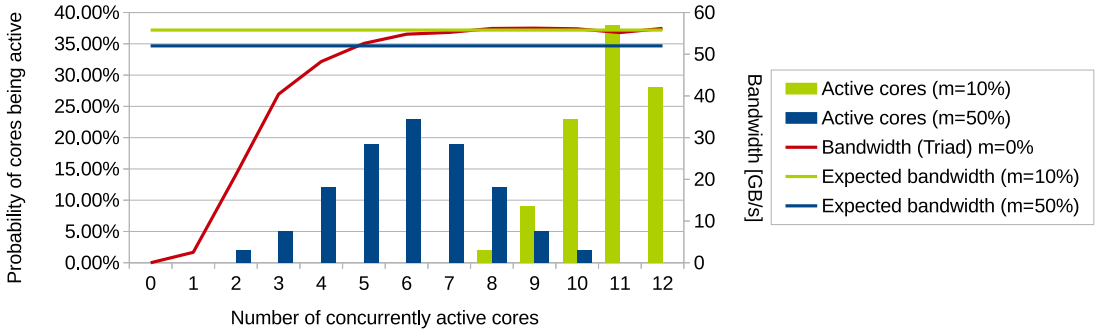


Figure 4.28: Binomial distribution of number of active cores on a system with 12 cores and different m. The expected bandwidth (green and blue lines) depends on the binomial distribution of active cores (which depends on m and the number of used cores) and the achievable bandwidth for the number of active cores. The depicted bandwidth information is based on results from the Intel Xeon E5-2680 v3 test system that are detailed in Section 4.4.4.

4.6 Conclusion

In this chapter, I showed that the hardware dependent parameters of energy efficiency models differ highly between processor architectures. P-states have different scopes and can influence the selection of other parameters like the uncore frequency. With the Haswell-EP processor generation, the average latency for changing P-states increased significantly by an order of magnitude. In general, C-state latencies have decreased over time. The exact time to re-enable a processor core depends significantly on the core that triggers the enabling and on the P-state of the processors. The T-states behavior of Intel processors also changed with the different hardware architectures with a significant influence on the used mechanisms, and the expected latency. This section wraps-up essential findings of this chapter.

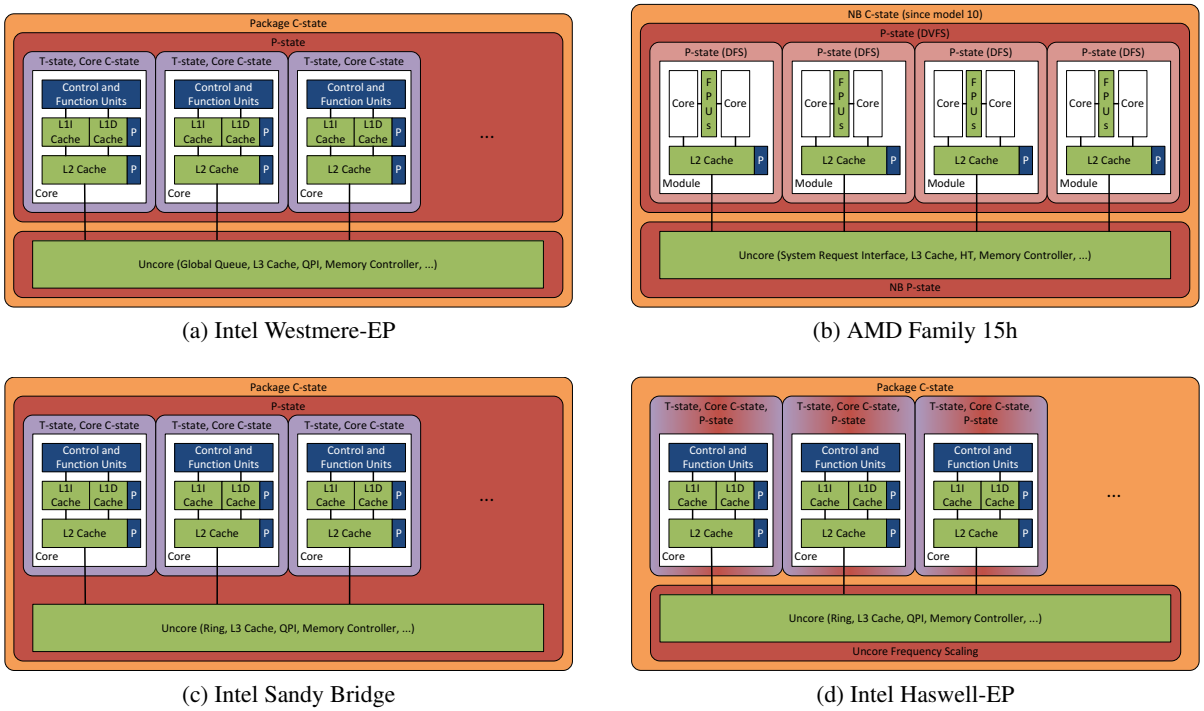
The scope of a power saving mechanism is a crucial information when considering a tuning for energy efficiency. While some mechanisms can be used at per-core scale, others are implemented per processor. Figure 4.29 illustrates the different scopes for power saving mechanisms on the used processors. The Haswell and Skylake desktop processors behave like the Intel Haswell-EP processor (Figure 4.29d), except that all cores share a common core P-state.

The access costs for P-states depend on the software mechanism used to toggle the respective MSRs. However, the fastest interface for such an access is not designed to be used from user-space and can thus not be used for energy efficiency optimizations. This interface is also the only way to use T-states. In Section 6.1, I describe and implement an interface to enable a safe access to the underlying MSRs.

The latency for enabling and switching ACPI states differs between the examined architectures. Typically, P-states are changed 10 to 100 µs after the request has been issued. Lowering the frequency is generally faster than increasing it, and the latency increases with the difference between source and target frequency. On AMD architectures, this delay is almost instantaneous for lowering the frequency. On Haswell-EP processors, which support PCPS, an external mechanism switches core frequencies at a time interval of 500 µs. Latencies for leaving C-states sometimes violate the ACPI standard by reporting wrong information to the operating system. In general, the latencies decreased over multiple processor generations. On Intel processors, the latencies for leaving C1 are between 0 and 2 µs, for C3 they are in the range of 10 to 35µs. Leaving a C6 state takes between 20 and 60 µs. Package C-states increase the latency by up to 20 µs. AMD Family 15h server processors have a significantly longer latency for leaving C1, which is up to 11 µs, while C6 states have a latency of up to 95 µs.

T-states are only supported by Intel. On newer architectures (starting with Haswell), they are enabled and disabled instantaneously after writing to the register. On previous architectures, the latency for enabling is 12.5 µs. The time for disabling T-states depends on the timing of the write operation to the MSR in relation to the clock modulation signal on newer architectures. On older architectures, the latency is 17 µs.

It is not always beneficial to use ACPI states for processor cores while the performance is limited by an external resource. If the uncore frequency is determined by the core P-states, the bandwidth to shared resources is reduced since I/O and memory controller are also slowed down. An independent uncore performance, however, does not guarantee that the achievable memory bandwidth remains constant when using power saving mechanisms. If the on-chip buffers are not designed for such scenarios, the main memory performance can also decrease. As soon as the size of buffers is able to hide the memory access latencies, a further reduction of core frequencies also decreases the achievable memory bandwidth. Concurrency throttling by using C-states can be an alternative that works on all architectures but can not be used for all programming paradigms. T-states can also be used, but the exact achievable performance is unknown, since the clock modulation signal is not synchronized.



(a) Intel Westmere-EP

(b) AMD Family 15h

(c) Intel Sandy Bridge

(d) Intel Haswell-EP

Figure 4.29: Power saving mechanisms within processors at different scopes

5 A Software Concept for Performance and Energy Efficiency Tools

> *"It states that the more similar two objects are, the greater the sympathetic link. The greater the link, the more easily they influence each other."*, Abenthy
> **The Name of the Wind** by Patrick Rothfuss

In Chapter 3, I described a model that can be used to determine energy-efficient configuration states of a hardware and software environment. I detailed how hardware dependent parameters for this model can be measured and presented hardware parameters for the test systems listed in Appendix A in Chapter 4. The remaining topic, which I address in this chapter, includes software parameters like region and function information, throughput p and power consumption P. The latter two factors depend on the instruction mix of the software and how the hardware is able to process this instruction mix. For example, if there is a significant number of stalls, throughput and power consumption decrease. Such software parameters should be available for online and offline tuning (see Section 2.5.4) in order to enable a common infrastructure for both tuning approaches. To increase the applicability of the solution, the infrastructure that provides this information should not be targeted at one specific tool or a specific architecture but be extendable.
In this chapter, I present a concept for such an infrastructure. I define the used terminology in Section 5.1. Based on this, I describe the parts of the monitoring infrastructure in Section 5.2. I show how this concept relates to existing performance measurement tools, tuning tools, and infrastructures in Section 5.3. In Section 5.4, I lay out which modifications are necessary to enable existing performance measurement infrastructures to change the hardware and software environment for energy efficiency purposes. In the following Chapter 6, I describe how I used this concept to extend existing tools to facilitate the measurement and tuning of a program's energy efficiency.

5.1 Basic Definitions

Each computing system consists of multiple hardware and software elements that can be monitored. I call one such element an *observable element* e. Observable elements (or in short: elements) can be distinguished into *hardware and software elements* (e_h and e_s). Hardware elements can be further distinguished into active (e_{h_a}) and passive hardware elements (e_{h_p}). Active hardware elements are used to execute software elements, passive elements only support the execution. For example, a processor core can be considered an active hardware element, a cache is a passive element. Elements can be grouped into *element groups* E, where $\bar{E}$ includes all observable elements of a computing system. Element groups can be used to describe a context between its member elements. Multiple threads (each representing an element), for example, can belong to a process (which is represented by an element group). However, single elements can also form an element group E^1. This notation can be used to generalize rules for sets of single and multiple observable elements. In Figure 5.1, I depict an exemplary single core system and a set of useful hardware and software observable elements.
I refer to a *monitored* element group as a *workload* ω. Usually, a workload consists of both, hardware and software elements. An element group can have different *properties* π, which define its internal state and can be monitored or modeled. Such properties could, for example, be the function that is currently executed, the contents of a register, or the current operating frequency of a hardware device. Based on its nature, each property has a specific value range $V = \{v\}$, which can be continuous or discrete. The content of a 64-bit integer register, for example, would be translated to a natural number n with $0 \leq n < 2^{64}$. If the values of a property can be described by numbers, I call it a *metric*. The values of properties

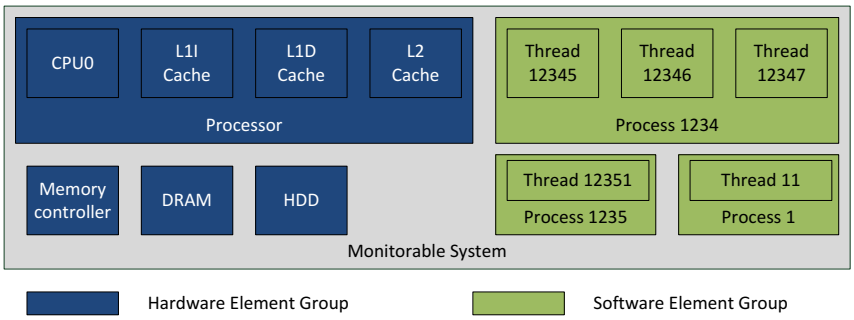


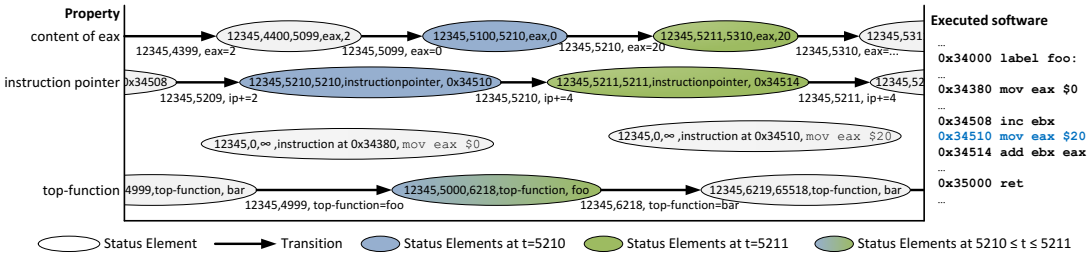
Figure 5.1: Example of observable software and hardware element groups in a single core processor system with two cache levels and split level 1 cache.

change over time, e.g, when a write on a register changes the register's content, a new value is set. I call a specific setting of a property over a specific time interval $[t_s, t_e]$ a *status element* $s = (E, t_s, t_e, \pi, v)$. The changes of properties' values are called *transitions* $x = (E, t_n, \pi, v, E_x), x \in X(E)$, where the element group E changes its property π at time t to a new value v, due to the influence of element group E_x. In the most simple case, E_x matches E. However, properties can also change due to the activity of other element groups. Transitions are initiated whenever a certain *transition rule* $r \in R(E)$ applies to the status of the computer system. These rules can be explicit (e.g., after issuing `mov r12 $20`, the content of register r12 will be 20), implicit (e.g., when an instruction is executed, the instruction pointer increases concurrently), or uncertain (e.g., how does the temperature of the hardware change). Transition rules are defined by the program and its loaded shared libraries in the form of instructions that are to be executed: by the ISA, which defines how the instructions should be interpreted, by the underlying hardware, which defines how the processing of instructions is implemented, and by the operating system, which defines external influences to the hardware/software interaction. Transitions are issued by specific element groups, but can also affect other element groups (e.g., a memory access by a processor core).

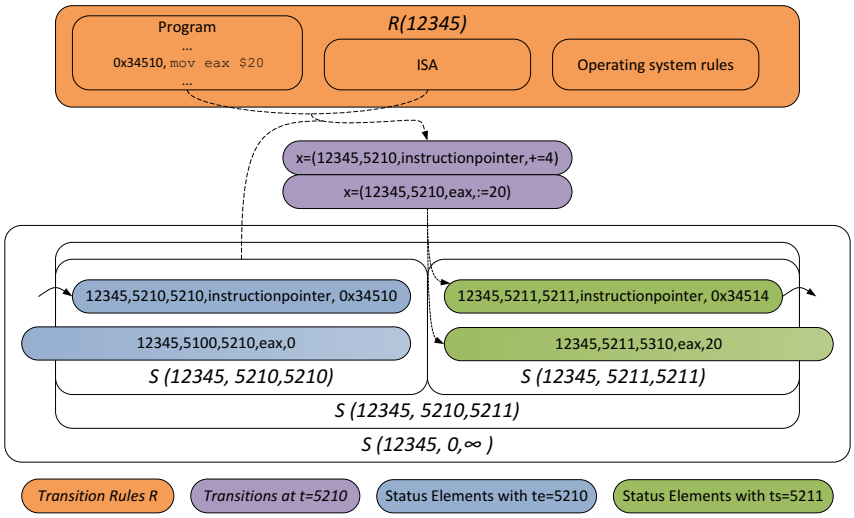
Within observable elements time can be considered to progress discretely, not continuously: hardware elements change their properties with every cycle, software elements with every instruction. I define a minimal transition from one point in time to the next one as a *time step* $t \in T(E)$. Transitions in multi-element hardware groups depend on their internal clock mechanisms. To cover all changes in the group, a temporal resolution that covers all elements of the group has to be introduced. Even though pipelining, out-of-order mechanisms, and speculative execution can obscure the exact status elements at a specific point in time, on a coarser scale the model is still valid. In order to enforce a consistent view over all status elements at a specific time step, special instructions can be used to synchronize the instruction stream or memory operations [Pao10].

An example for status elements and transitions is depicted in Figure 5.2. The figure shows a small section of all the status elements over the execution of a single element software group. The depicted thread executes the function `foo` that is called by the function `bar`. At a specific point in time (t=5210), the thread executes the instruction at address `0x34510`. The instruction at this address reads as `mov eax $20`. The transition rules given by the ISA define that the value of 20 should be written to the register eax and the instruction pointer should be increased by 4. Thus, after the instruction is issued, the instruction pointer is increased by four and the content of register eax changes. Additionally, the number of executed instructions (the time step) is incremented.

Multiple status elements can be encapsulated in a *status* $S(E, t_s, t_e)$. An example for such a status is depicted in Figure 5.2b. Here, elements of *S(Thread 12345, 5210, 5210)* are marked blue and *S(Thread 12345, 5211,5211)* green. *S(Thread 12345, 5210, 5211)* contains all of the marked elements. If the transition rules and a specific status at a certain point in time is known, follow-up statuses can be determined. However, statuses can also contain information over a time span. In this case, a behavior of the workload can be expressed either by the transitions within the status or by representative samples at different time stamps.



(a) The value of a property changes over time when transitions change it.



(b) Transitions are a result from the previous status (at t=5210) and the transition rules that are defined by hardware, software, and operating system.

Figure 5.2: Example of status elements and transitions of a single software element group. While the time steps (executed instructions) increase, different transitions for different properties are executed. E.g., with every executed instruction, the instruction pointer is changed. Some status elements do not change over time (e.g., the instructions at a specific address in non-self-modifying software), some change at every time step (e.g., the instruction pointer), some at a coarser time scale (e.g., the current function or the content of a specific register). At a specific point in time status elements for several properties are valid. These form the status at this point in time.

I now define how software and hardware groups can be combined by introducing scheduling information. Such information is important to map status elements between element groups of a typical workload that monitors software that is executed on hardware elements. A thread E^1_s is scheduled on an active device $E^1_{h_a}$ in a specific time interval $[t_s, t_e]$ and executes a number of instructions $[i_s, i_e]$. While the thread is executed, the statuses of both elements (thread and device) change. The software itself is not able to process from one status to another, but the active hardware device drives the status of both. In order to map the status of a hardware element group $E^1_{h_a}$ to an element group with software and hardware elements, I define the *scheduling status* S_{sched} as a set of specific status elements with the property *scheduling*. The scheduling status enables me to define a group of software and hardware elements and describe their interaction. In the simplest case, this group consists of exactly one active hardware element and one software element. Such a group is denoted as $E^1_{sh_a}$ and described in Equation 5.1.

$$\begin{aligned} S(E^1_{sh_a}, t_s, t_e) = S(E^1_{h_a}, t_s, t_e) \cup S(E^1_s, i_s, i_e) \mid \\ (E^1_{h_a}, t_s, t_s, \text{scheduling}, (E^1_s, i_s, i_s)) \in S(E^1_{h_a}, t_s, t_e), \\ (E^1_{h_a}, t_e, t_e, \text{scheduling}, (E^1_s, i_e, i_e)) \in S(E^1_{h_a}, t_s, t_e), \\ \forall t \exists! i | t_s \leq t \leq t_e, i_s \leq i \leq i_e : (E^1_{h_a}, t, t, \text{scheduling}, (E^1_s, i, i)) \in S(E^1_{h_a}, t_s, t_e) \end{aligned} \tag{5.1}$$

However, the hardware element in $E^1_{sh_a}$ has to be independent from other elements. Thus, when simultaneous multithreading is used, a property of a hardware thread cannot be related directly to a specific software thread. To apply a status element of a hardware group with multiple elements E^+_h to a group of software elements E^+_s, each active device of the former has to execute one element of the latter ($|E_s| = |E_{h_a}|$) at all times and the correct scheduling has to be known. To match an uneven number of software and hardware elements, additional active hardware elements can execute predictable software elements (e.g., an OS-scheduling loop). This increases the number of known software elements. Alternatively, the unused hardware devices can be stopped (e.g., by ACPI Power States), which reduces the number of active hardware elements. The resulting status is described in Equation 5.2.

$$
\begin{aligned}
&\overbrace{S(E^+_{sh}, t_s, t_e)}^{\text{Combined status}} = \overbrace{S(E^+_h, t_s, t_e) \cup S(E^+_s, i_s, i_e)}^{\text{holds hardware and software elements}} \mid \\
&\overbrace{\forall e_{h_a}, t \exists! e_s, i | e_{h_a} \in E^+_h \cap \bar{E}_{h_a}, t_s \leq t \leq t_e, e_s \in E_s, i_s \leq i \leq i_e}^{\text{For all active hardware elements at all times there is exactly one known monitored software element}} : \\
&\overbrace{(\{e_{h_a}\}, t, t, \text{scheduling}, \{e_s\}, i, i)) \in S_{sched}(E^+_h, t_s, t_e)}^{\text{... that executes a specific instruction and is scheduled at the time.}} \\
&\wedge \overbrace{\forall E, t \mid E \subseteq E^+_h, t_s \leq t \leq t_e : \nexists e_{extern} \in \bar{E} \setminus E^+_h : (E, t, \pi, v, e_{extern}) \in X(E)}^{\text{Also, there is no unmonitored active hardware element that influences any monitored hardware element group.}}
\end{aligned}
\tag{5.2}
$$

Re-scheduling of threads can lead to significant performance degradation as, for example, cache contents cannot be re-used and memory can be assigned to the wrong NUMA-node. Therefore, most HPC paradigms and the underlying software stack support the pinning of threads and processes to specific cores. The batch system SLURM, for example, pins MPI ranks to allocated cores in a cluster per default. The individual processes are now only interrupted when the operating system schedules additional work to the cores. However, this should only happen sporadically as it can lead to significant imbalances in parallel programs [PKP03]. If there is no change in scheduling over time, the complexity of the scheduling information reduces significantly. If it can be assumed that there is no influence on the hardware devices from any device that is not part of E^+_{sh} within the time interval, one can say that the behavior of the hardware elements in E^+_{sh} depends directly on its previous status, the scheduling information and the transition rules. However, with an increased time frame, the probability of transitions $X(E^+_{hs})$ within this time frame increases and the influence of the previous status decreases. Additionally some hardware and software functionality can create an initial "clean" state (e.g., via cache flushes).

5.2 Description of Measurement and Tuning Tools

In the previous section, I described how the status of a workload changes over time. Now, I will point out how this influences the way a status is measured. To do so, I describe challenges when processing statuses of workloads and the implication on monitoring and tuning tools. I further describe how a monitoring tool can be defined and how the monitored information matches the internal status and transitions.
The complete status of a workload has numerous status elements, several of which are hidden due to a lack of documentation. This lack of documentation can be fixed on the software side, e.g., by disassembling a binary. On the hardware side this issue cannot be resolved as easily, since vendors do not disclose all information of internal processing. Another problem of a detailed status is the amount of data that would have to be processed and the involved processing overhead. Thus, in performance measurement tools most of the status is not reflected, but only specific properties that are deemed necessary for understanding the underlying mechanisms. Some performance measurement and tuning tools are based on instrumentation (see Section 2.3). The kind of instrumentation that is used defines which regions should be available in the data representation layer and thus, in the analysis. Here, a trade-off between performance perturbation and a higher number of status elements has to be considered. The same decision

has to be made when deciding about the sampling frequency if the performance measurement or tuning tool relies on sampling. An additional reduction in overhead can be achieved by using *statistical status elements* $\tilde{s} = (E, f, t_s, t_e, \pi, v)$. These use a statistical description, based on a statistical function f of how a specific property of an element group changes over time. Processor vendors support this kind of analysis with Performance Monitoring Units (PMUs), which provide accumulator registers for different hardware events [Int15a, Chapter 19],[Adv13, Chapter 2.7]. In order to maintain the assignment of hardware elements to software elements, operating systems like Linux save and restore these register contents when re-scheduling threads. Shared hardware elements such as uncores cannot be mapped directly to software element groups. However, to match these, Equation 5.2 can be used.

When the workload is to be observed, its execution has to be interrupted by a processing infrastructure. This infrastructure, depicted in Figure 5.3, consists of at least three parts: *front ends*, *integration*, and *back ends*. Each front end uses one or more *event generators* to interrupt the workload. Such an interrupt can be issued by every hardware or software element of the workload. Usually, hardware interrupts are used for sampling, software interrupts for instrumentation. One example of a front end is the instrumentation of MPI functions via the MPI Profiling Interface (PMPI) [mpi15, Section 14.2]. Here, the interrupted functions represent the event generators. When such an interrupt occurs, the respective front end creates an initial monitoring status $\tilde{S}(E_s^1, t_n, t_n)$ that is forwarded to the integration. The elements within $\tilde{S}$ are defined by the event generators. For the PMPI example, the passed status elements can include the function parameters that are passed to the MPI library. However, the status created at the front end does not necessarily have to provide any status element but can also create an empty status ($\tilde{S}(E, t_n, t_n) = \emptyset$). The integration part can use additional status element collectors to define the status more clearly by adding more elements $s(E)$ or $\tilde{s}(E)$. Captured information does not necessarily only relate to the current time t_n, but can also relate back in time ($t_s < t_n \wedge t_e = t_n$) or relate to the future $t_s = t_n \wedge t_n < t_e$. Afterwards, the resulting monitored status can be processed by the back ends. Based on current and previous statuses, each back end executes specific processing *actions* like writing some status information to a buffer or a file (measurement), or changing the hardware or software environment (tuning). Information from previous statuses can be buffered in the back ends or the integration layer. This is especially important for status elements that relate to a time interval $[t_s, t_e]$ to mark their respective end or beginning and reduce measurement and recording overhead. The action of a back end can also be used as another tools' event generator or *status element collector*.

5.3 Existing Tools and Tools Infrastructures

I introduced performance measurement and energy efficiency tuning tools in Section 2.3 and Section 2.7, respectively. In this section, I show how existing tools fit into the concept. Since the group of perfor-

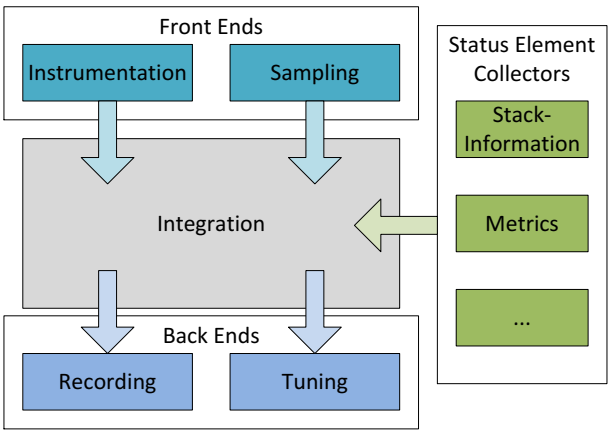


Figure 5.3: Measurement and tuning tools, general overview

mance measurement tools is more homogeneous with only two kinds of back ends (profiles and log files) and two possible front end types (instrumentation and sampling), I only describe two of them: Score-P, which is targeted at HPC workloads, and perf, which targets compute node monitoring. Afterwards, I classify a number of possible energy efficiency optimization tools and show that the concept can also be applied to tools infrastructures by providing examples.

Score-P [KRaM+12] is a scalable performance measurement infrastructure with a significant number of front ends and status element collectors. The single components are separated in the following way: in the Score-P terminology, front ends are called "adapters". These utilize instrumentation interfaces and tools for compilers and established parallelization paradigms like MPI, OpenMP, CUDA, SHMEM, and Pthreads, but also sampling. The front ends call Score-P internal functions that use "services" (representing status element collectors) to provide more status elements to the back ends that are called "substrates". The available back ends include performance measurement (profiling and tracing) and an interface for online accesses that is used by the Periscope tools. At the beginning of my work, all components of the infrastructure were defined statically when compiling Score-P. Thus, they were not extendable. Another performance measurement tool that can be used at compute node scale is the profiling and tracing capability of the perf userspace tools [Wea13], which is executed by issuing the `perf record` command. Here, front end, integration, back end, and status element collectors are also firmly integrated and cannot be exchanged easily. The front ends of the tool reside in the perf_event kernel subsystem and provide sampling and instrumentation event generators. A limited number of status element collectors is available that attributes additional information to the individual statuses. This includes scheduling and timing information as well as the retrieval of a call-path. The back end dumps the recorded data to a log file that can be analyzed later with profiling or tracing tools. Other performance measurement tools listed in Section 2.3 can be described analogously.

Tuning tools that are discussed in Section 2.7.2 can be described by the introduced nomenclature as well. The Linux ondemand cpufreq governor [PS06] changes P-states according to the idle-time in the last time-frame. It does so by using a sampling based front end. At regular intervals, the monitoring routine captures the usage information of the hardware thread as status element and changes the P-state accordingly, which represents a tuning back end. These parts are integrated in such a way that it is not trivial to exchange the single components. The Green Governors [SKK11] and the pe-governor [SH11] have the same front end (sampling) and back end (P-state changes), but use Performance Monitoring Counters as status element collector to base the frequency decision on memory boundedness.

ACPI C-state decisions in Linux systems are made by the cpuidle governor [PLB07]. This governor is called by the scheduling algorithm if there is no thread that waits for processing. Thus, the cpuidle front end is based on instrumentation. The governor then checks how long it expects the hardware thread to be idle, using different heuristics. It then applies the idle routine of the selected C-state as back end. It also monitors the quality of its decision and stores status elements for further reference.

Adagio [RLdS+09] uses an instrumentation via PMPI as front end and an algorithm to increase the energy efficiency of the computing system as back end. The status element collectors provide a call-stack identifier that is retrieved by creating a hash on the call-stacks return addresses and a timestamp. Statuses are stored internally to match events with a common call-stack identifier. Likewise, LaBaTa [Mül13] and Green Queue [TLP+12] also use instrumentation of MPI programs as front end and a back end that changes the frequency according to the imbalance. However, the integration of LaBaTa exchanges imbalance information with other MPI ranks to get the current imbalance. Green Queue uses an existing imbalance profile as status element collector.

In addition to performance measurement and tuning tools, there are generic infrastructures that can already be used for multiple purposes like measurement, debugging, and tuning. In [LOW96], Ludwig et al. describe the On-line Monitoring Interface Specification (OMIS). OMIS is an interface that can be used by tools to access monitored threads and processes. It initially targets MPI and Parallel Virtual Machine (PVM) front ends but is designed to also support *"monitor extensions [that] are used when there are additional system components to be observed"*. Thus, the number of available front ends can be increased by implementing such an extension. A monitoring extension can also substitute a status

element collector. OMIS is furthermore extendable by *"distributed tool extensions"* and *"distributed tool components"*, according to [LOW96, Section 5.5]. These enable the filtering of front ends and the implementation of back ends that can be triggered from the respective front ends. On initialization, the tools register for the back ends that should be activated. Thus, measurement overhead is only introduced from front ends that are used by at least one tool. Ludwig et al. describe that their approach can be used for *"debuggers, performance analyzers, program flow visualizers, checkpointing facilities, and load balancing"* [LOW96, Section 4]. Hilbrich et al. describe the Generic Tools Infrastructure (GTI) and apply it to an MPI front end with a function profiler and a correctness check back end in [HMdS+12]. GTI can be understood as the integration part for distributed measurement or tuning tools. The description of a tool consists of *"analyses (actions) that the tool uses to provide its service [,] functions (events) to intercept"*, *"available communication modules and drivers"*, and *"the layout of a tool instance"* [HMdS+12, Section III A]. The *"functions"* define the front ends of the different instances based on the tool that is used. The *"analyses"* represent back ends, which can also be used as front ends for other analysis steps. The remaining GTI parts enable the communication between individual processing instances, and build the integration. Each of the instances can be seen as a measurement tool with a set of front ends and back ends. Another approach of a distributed tool infrastructure is TAUoverMRNet [NMM+10]. Here, a central *"Front-End"* controls the individual *"Back-Ends"* that later create events that are passed to the Front-End. In the terminology that I introduced in Section 5.2, a TAUoverMRNet Front-End, represents a back end and vice versa. The single event generators are given by the TAU [SM06] instrumentation. Periscope [BPG10] describes a distributed tool infrastructure where the back end is able to configure status element collectors and front ends at runtime. In the current implementation, Periscope uses the Score-P [KRaM+12] infrastructure and thereby Score-P's front ends and status element collectors. The Periscope Tuning Framework [GCB15] uses this infrastructure to tune the runtime and the energy efficiency of parallel programs.

5.4 Implications for Extending Existing Measurement Infrastructures

In the previous sections, I presented a concept to describe performance and energy measurement and tuning tools. As I discussed, energy efficiency tuning tools have only a limited number of front ends, most often just a single type of instrumentation or sampling. Thus, I propose to extend performance monitoring tools to be able to capture power consumption information and tune the system configuration to increase the energy efficiency. This implies that such a framework should be able to handle new status element collectors and back ends. In this section, I describe how existing infrastructures should be extended to use the different available power information sources that are discussed in Section 2.4 as status element collectors and how a back end interface should be designed to enable online, and offline tuning on local and global scale for energy efficiency tuning (see Section 2.5.4).

5.4.1 Status Element Collectors

Status element collectors provide contextual information at specific points in time, e.g., at each event. The properties that are described by the provided status elements differ by the described element group, i.e., the spatial scope, the value range of the property, and the temporal scope that does not necessarily have to match the event rate of the front ends.

Spatial Scope

In Section 5.1, I declared that element groups have properties that define their internal status. Performance measurement tools focus on a specific subset of the available observable elements, mostly software elements like threads and processes. For the model that I described in Chapter 3, hardware related and software related information has to be combined to be able to evaluate and predict the energy

efficiency of programs and functions thereof on specific computing systems. Thus, status elements of different groups have to be merged. As I described in Section 5.1, the merging of different element groups is based on scheduling information of the participating active hardware elements. On Linux systems, the scheduling information is partially transparent for the participating software elements. Each software thread is able to gather its current hardware thread via `sched_getcpu()` and the affinity mask via `sched_getaffinity()`. Additionally, the perf subsystem can be used to capture the time that the thread is actually scheduled. However, the concrete scheduling events cannot be measured without privileged access, as I described in [STHI11]. The available scheduling information maps one software thread to one hardware thread. To map properties of shared resources like uncores or processors, tools need topology information about the hardware, which is, for example, provided by the `hwloc` library [BCOM+10]. This topology information can be used to describe hardware element groups based on the hardware architecture. This information is not collected by most performance measurement tools and would require appropriate support in the resulting profiles or log files. Nevertheless, this information is crucial to map status elements of different element groups. For example, a hardware element group E_h^+ that includes all hardware elements of a specific processor can be used to bring together properties of all of its hardware elements. With the addition of all software elements that are scheduled to the active hardware elements of the mentioned group E_h^+ at the monitoring time interval, it is possible to describe the software/hardware interaction. For example, there is a workload with exactly two active hardware elements $core_1$ and $core_2$, which share a passive hardware element $cache$. In a specific time interval $[t_s, t_e]$, the scheduling information for $\{core_1\}$ and $\{core_2\}$ specifies that $core_1$ executes a specific software thread $thread_1$ and $core_2$ executes $thread_2$. As long as $cache$ is not influenced by any other element that is not part of the monitored workload, the monitored behavior of $cache$ is the direct result of $thread_1$ and $thread_2$. However, it is not possible to attribute changes in $\{cache\}$'s properties to a specific thread. Likewise, an element group of hardware threads of a processor core and the scheduled software threads can be used to describe the status of a core that executes the software. However, as I said, performance monitoring tools often neither support compute node topology nor detailed scheduling information. Still, some information is available. This includes the compute node on which single software elements are executed, and inherently a global perspective that includes a list of the monitored software elements and the used compute nodes. Thus, in addition to different software element groups, only two additional scopes can be considered as long as the node topology and the scheduling information is not available for analysis. These are the *per-host* scope, which includes the monitored software and hardware elements of a single compute node and the *global* scope, which includes all monitored software elements and the used hardware elements of the analyzed workload.

Value Range

Aside from the affected element group, each property has a specific value range, which describes the different values that can be assigned to it. Based on the type of the property, this value range can differ. In this thesis, I focus on such metrics where values can be mapped to numbers. Existing performance measurement tools already support metrics like PAPI. Thus, available definitions can be reused to supply additional information to the existing data stream that is recorded. This enables the analysis tools to read and plot for example power consumption information alongside performance information, which would be a first step of analyzing the energy efficiency of a program. Metrics that are supported by performance measurement and analysis tools use specific data types, i.e., signed and unsigned integers or floating point numbers, which are standardized. Computation of these data types is executed in hardware and can thus be deemed to be performant. Values that do not meet an internal data format cannot be recorded (e.g., 2^{64} cannot be represented by a 64 bit integer).

Information Type

Depending on the type of information that is represented by a collected status element, the validity of the information can be assigned to the region before the property is measured (*backward looking*), to the

time when it is measured (*instantaneous*), or to the time after it is measured (*forward looking*). Typical backward looking status elements are statistical status element and represent a statistical function of transitions or a filtered value of a property over a specific time frame. One example is the recording of PMCs. Here, the exact information (e.g., when did the individual cache access happen) is condensed into a single value (e.g., how many cache accesses happened since the last reading). Instantaneous status elements are only valid at one point in time. Thus, they cannot be mapped to a computing region as it would be usual for performance monitoring tools. However, more specific assumptions about the property's behavior and the quality of the measurement can be used to combine multiple instantaneous status elements of the same property to a statistical status element that can be related to a region. This is, for example, done when creating profiles from sampled functions. Forward looking status elements describe changes of properties. Here, a transition of the property is captured and the new status element is recorded. One example for this is the recording of instrumented functions, where the function is pushed to or popped from the recorded stack when the function is entered or exited, respectively.
The three different types can be converted into each other. However, such a conversion is subject to certain restrictions if the status elements are to be supplied to the performance monitoring tool in ascending order of their time stamps. If a backward looking status element is to be converted to a forward looking one, all status elements of other properties must be buffered. For example, a backward looking status element $s_i = (E, t_{s_i}, t_{e_i}, \pi, v_i)$ can be recorded at t_{e_i}, but shall be recorded at t_{s_i}. Here, the status of all recorded events, including the collected status elements at the time t, where $t \geq t_{s_i}$ and $t \leq t_{e_i}$, must be buffered. At t_{e_i}, the now forward looking status element can be written and the buffered data can be flushed with t_{s_i} as a time stamp. However, such a buffering would require a significant amount of memory if the event rate of the front ends is higher than the update rate of the property π. To translate a forward looking status element to a backward looking one, only the status element has to be buffered until its successor is available. As I explained earlier, instantaneous status elements cannot be converted without additional assumptions. However, forward looking ones that represent a transition can be translated into instantaneous ones, as at each point in time, the current value of the property is valid. However, it should be noted that statistical status elements cannot be converted to instantaneous values.

Temporal Scope

Since properties are changed at different rates, it is not necessarily reasonable to collect them whenever a front end interrupts the workload due to a monitored event. Thus, status element collectors should be able to provide their data independently, or *asynchronously*. Statistical status elements like PAPI metrics are related to a region and would thus be *synchronous* to the status element provided by the front ends. In contrast, asynchronous status elements can still be related to each other, depending on the information that is provided. For example, the power consumption of a compute node depends on the number of active processors and processor cores, the applied frequencies, and the executed software. Thus, appropriate power consumption status elements can be related to performance measurement information, which provide the number of executed threads, the scheduling information, and the number of executed cycles per second. For example, over a certain time frame $[t_{p_s} - t_{p_e}]$, the average power consumption of 250 Watt is measured. The monitored workload ω consists of a set of executed threads $E_s \subset \omega \wedge E_s \subseteq \bar{E}_s$ and a set of all processor cores of the monitored compute node $E_{h_a} \subset \omega \wedge E_{h_a} = \bar{E}_{h_a}$. The power consumption is measured for the whole compute node $\bar{E}_h$. If each core is associated with exactly one monitored thread in the respective time frame $\forall core \in E_{h_a} : \exists!$*scheduled thread* $\in E_s, s(\{core\}, t_s, t_e, scheduling,$ *scheduled thread*$), t_s \leq t_{p_s} \wedge t_e \geq t_{p_e}$ and if each thread executes the same function foo in the respective time frame $\forall$*scheduled thread* : $s(\{$*scheduled thread*$\}, t_s, t_e,$ executed function, $foo), t_s \leq t_{p_s} \wedge t_e \geq t_{p_e}$ it can be assumed that the average power consumption is a result of the average impact of the executed function foo, the applied software and hardware configuration, and the previous hardware status. However, if multiple functions are executed within the time frame $[t_{p_s} - t_{p_e}]$, no such assumption can be made. Thus, status elements that are provided asynchronously can be useful for a later analysis, but have inherent restrictions.

Summary

Properties can have different spatial scopes, value ranges, information types, and temporal scopes. These have to be supported by monitoring and tuning tools to be able to capture information properly. In the previous paragraphs, I defined the following issues that have to be addressed:

- The spatial scope of a captured status element can be any element group. Since most performance measurement tools focus on software elements and do not provide detailed hardware topology descriptions, the suggested element groups are *per thread*, *per process*, *per compute node*, and *global*.
- The value range of a property can be any set possible. A performance measurement tool should support at least those that can be represented by a *standardized data type*, i.e., integers and IEEE floating point numbers.
- The information type can be *forward looking* (for captured transitions), *instantaneous* (for instantaneous measurements) or *backward looking* (for statistical status elements). All of these types should be supported to be able to capture the associated information correctly without having to convert these into each other to guarantee a correct order of the captured information.
- The temporal scope of captured status elements can differ depending on the provided information and the respective data source. A performance measurement tool should be able to capture these without being restricted to an event rate that is dictated by the respective front ends.

5.4.2 Back Ends

Back ends consume the statuses that the integration passes to them whenever an event interrupts the monitored workload. Based on the type of targeted action, a back end needs specific status elements to be available. For example, a back end that records an instrumentation based runtime profile needs to know whether an Enter or an Exit event has been triggered, the current time, information about the function that is entered or exited (e.g., the name), and, if the recorded workload is parallelized, the software element that interrupted the workload. If one of these status elements is missing, the back end cannot fulfill its purpose. Thus, back ends must be able to make sure that all their preconditions are fulfilled. For specialized tools such a demand is easy to fulfill, since the objectives of front ends, status element collectors, integration, and back ends are defined in advance. For frameworks that support different types of back ends, this is not always the case. Therefore, in tool suites like Periscope and OMIS, back ends are able to set-up respective front ends and status element collectors. In this section, I describe what kind of changes must be applied to a performance measurement tool to make it applicable for different types of back ends. I distinguish between *local* and *global* processing of the received events, and between *online* and *offline* tuning. I defined these terms in Section 2.5.4.

A local processing does not allow any communication between the resources that are responsible for the individual observable elements. Thus, these mechanisms are inherently scalable. Examples for a local measurement are all HPC performance measurement suites described in Section 2.3. However, in the initialization and the finalization phase of these tools, collective synchronizations are used to allow a global view on the application. Still, in the measurement phase, the processing is local.

The processed events can be used for online tuning if the available statuses provide all the status elements that are needed for the underlying optimization model. For example, the ondemand governor processes events locally, it collects the load of the processor over the last time frame and the currently set frequency as status elements, and, based on these, a simple model predicts a more suitable frequency, which should be applied for the next time frame. Likewise, Adagio uses the PMPI Interface, collects the stack id and the current time stamp as status elements, and, based on its model, predicts the slack of the assumed following region and applies the respective frequency. Even though such tuning mechanisms can be used to predict local optima, a global optimum cannot be achieved, since the single resources are not allowed to communicate. Thus, local tuning can lead to a lower global performance.

Models can also be applied offline. Here, one or multiple measurements of a workload that is to be tuned are recorded. The model uses these measurements as input parameters and determines configurations that would lead to an efficient execution of the workload. The resulting configuration decisions can be spread to the local monitoring resources at the initialization phase and can then be applied locally, even though the tuned element group can be global. This is, for example, done by Periscope Tuning Framework and Green Queue.
Alternatively to a local processing, a global processing of incoming events is also used by different optimization tools and infrastructures. Examples for a global processing within a compute node are the tools strace, ltrace, and perf (introduced in Section 2.3). All of these use a single file to write their timelines to for all monitored threads and processes. To guarantee a correct processing of the incoming events, the tools use internal synchronization, which limits their scalability. An alternative are distributed tools infrastructures. GTI and TAUoverMRNet externalize the processing of the incoming events to monitoring processes, which are ordered hierarchically. Within this hierarchy, events can be filtered. This has two advantages. First, the processing overhead is externalized to other computing resources. Thus, the performance perturbation is significantly reduced. Second, the hierarchical filtering enables a better scaling. However, when applying these infrastructures, the resulting information is processed asynchronously to the captured events. Thus, the model I proposed in Chapter 3 cannot be applied. Tools like LaBaTa use sporadic global synchronizations to share local data. Therefore, statistical status elements of the global element group are available when processing information locally. Based on the used optimization model of the respective back end, such global information can significantly improve the quality of an optimization.

Summary

Back ends have different requirements concerning the events that interrupt the workload and on the passed status elements that they receive. Thus, one demand for a unified environment must be that the definition of status elements is available to the used back ends. This includes status elements that are received from front ends and from status element collectors. If this is the case, back ends can distinguish events and status elements that are relevant for their actions from others. Ideally, back ends are able to define front ends and status element collectors. However, if multiple back ends are active and the integration does not filter the events and status elements, the individual back ends have to implement such a filter.
Furthermore, back ends in a parallel workload should be able to communicate with each other to create a common global status information that can be used to increase the effectiveness of a possible tuning. To be able to flush collected information for an offline analysis, back ends must receive an event when the monitored workload finishes. Likewise, when the monitoring is initialized, an appropriate event enables back ends to read existing configuration variables that might stem from an offline analysis.

5.5 Conclusion

In this chapter, I have described a concept that can be used to combine measurement and tuning tools. I started by describing how software and hardware properties behave over time and how they can be mapped to each other. Then, I described single components of an infrastructure that can be used for measurement and tuning, and how the concept applies to existing tools. I close this chapter with a discussion on the implications of my concept and focus on two different types of components: status element collectors, which provide information about the measured software and hardware, and back ends, which use this information for measurement or tuning. In the next chapter, I describe how I extended existing performance measurement tools based on these findings.

6 Realization of Software Interfaces and Use Cases

> *There is something deeply satisfying in shaping something with your hands. Proper artificing is like a song made solid. It is an act of creation.*, Kvothe
> **The Wise Man's Fear** by Patrick Rothfuss

In the previous chapter, I described how performance measurement and energy efficiency optimizations both use the techniques of sampling and instrumentation. Furthermore, I showed that they rely on the same software parts: front ends, integration, back ends, and status element collectors. However, at the beginning of my thesis, none of the existing tools enabled users to integrate all the functionality of measurement and tuning within one infrastructure. This chapter describes how I designed and implemented interfaces to improve existing software in order to cover these aspects.

To implement the changes I suggested in the previous chapter, I used VampirTrace and its successor Score-P as a base, since they already support the common programming languages and parallelization paradigms. Both tools are described in Section 2.3. I separate the contents of this chapter according to the implemented aspects. In Section 6.1, I describe a Linux kernel module that provides access to hardware mechanisms (including those targeted at power saving) to user-space applications. This interface will be used in later sections to capture status elements and to enable energy efficiency tuning. I introduced this kernel module in [SM13]. Afterwards, I describe how I extended VampirTrace with an interface to easily add status element collectors that provide metrics in Section 6.2. This work has previously been presented in [STHI11] and has later been merged to Score-P. To extend the back end functionality of Score-P, I developed another plugin interface that is described in Section 6.3 and has been presented in [STI+17]. Additionally, I present examples to show how these interfaces can be used for performance and energy efficiency measurement purposes and energy efficiency optimization.

All of the presented software is available online under Open-Source licenses.

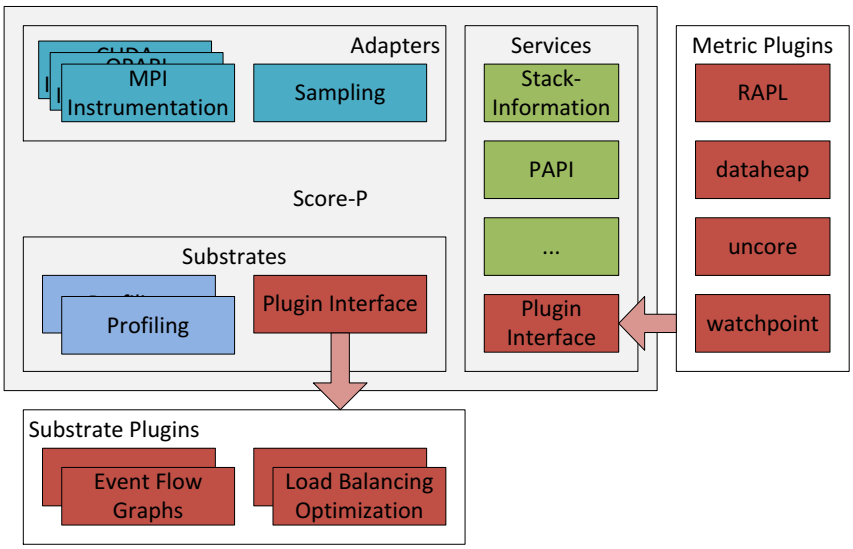


Figure 6.1: Overview of extensions made to Score-P in order to create an integrated recording and optimization framework for performance and energy efficiency. New interfaces that are based on this thesis are marked red. Other parts of the software are colored according to the general description depicted in Figure 5.3

6.1 Providing Access to x86 Hardware Mechanisms

Current x86 processors implement numerous optimization mechanisms that increase the throughput or energy efficiency. However, these optimization goals can be contradicting. The response time of a server is, for example, reduced significantly when processor power states (see Section 2.6.2) are not used. However, this would decrease the performance of single-threaded applications and increase the power-consumption of idling processor cores. To enable system vendors and administrators to optimize the processor for their design goals, processor vendors describe how to access and manipulate hardware mechanisms in processor manuals, e.g., [Int15a, Adv13]. While some of these settings can only be applied when the system is initialized (i.e., when the boot process is invoked and the BIOS manipulates the respective mechanisms), others can be changed at runtime. Usually, the manipulation is triggered by changing MSRs or CSRs. I distinguish the available parameters in these registers into three categories: configuration, tuning, and information. Configuration parameters have to be set at initialization time. When they are changed at runtime, the new setting might be ignored or the change can result in an instable system. This could happen, for example, when the System Call Target Address (see [Int15a, Table 35-2]) is changed. Tuning parameters can be changed at runtime and influence the behavior of the hardware element that correlates with the modified register. Information parameters are intended to be read and provide insight about the hardware component. In this work, I focus on the latter two types to enable runtime tuning and to access performance and energy efficiency information. Within a single register, these different parameter types can co-exist. Thus, any software that makes them available for manipulation has to support fine-grained access rights. This includes register white listing to define available registers that hold parameters, register masks to specify which parts of the register are allowed to be read or written, support for value ranges to make sure no invalid setting is applied to a parameter, and user-based access rights.

There are two different approaches to access tuning and information parameters under Linux. The first approach involves specialized kernel drivers for each distinct hardware feature. These kernel drivers provide a safe access from user space to a number of parameters. For example, processor frequencies can be set using the *cpufreq* kernel module, and PMUs can be read using the *perf_events* interface. However, only a limited number of these parameters is accessible via specialized interfaces. Thus, researchers have to develop own kernel modules to access features that are beneficial to them. For example, in [WDF+14], Wamhoff et al. implement a kernel driver that provides access to turbo settings of Intel processors to investigate its influence on performance and energy efficiency. Still, the vast majority of processor options is only accessible from user space via low-level interfaces, which are listed in Table 6.1. However, the access rights granularity is limited and can only comprise none or all registers of a hardware component. These underly additional constraints that make them difficult to use for non-privileged users. For example, binaries that read the *msr* kernel module have to be registered in the Linux capabilities subsystem [HM08] to have the `CAP_SYS_RAWIO` capability. To overcome these limitations, Walker and McFadden describe msr-safe in [WM16], which establishes another interface within the `/dev/` file system that can be used to white list specific regions of MSR registers. In an earlier publication, I described a kernel driver that enables users to access MSRs and CSRs while supporting fine-grained parameter granularity [SM13]. Table 6.2 lists main differences between the different approaches.

Table 6.1: Common Linux Interfaces for Register Access

	MSR registers	CSR registers
Kernel module/driver	msr	pci
File location	`/dev/cpu/*/msr`	`/proc/bus/pci/*/*`
General access	`pwrite/pread`	`pwrite/pread`
Library support	`libmsr`	`libpci`
Access rights	UNIX capabilities system	privileged access
Access granularity	per CPU	per device

Table 6.2: Kernel Modules to Safely Access MSRs and CSRs

	x86_adapt[SM13]	msr-safe [WM16]
Source	`https://git.io/v6nnc`	`https://git.io/v6nnB`
Support for MSRs/CSRs	✓/✓	✓/✓
Support for register white listing	✓	✓
Support for register masking	✓	✓
Value ranges	✓	✗
Reset	✓	✗
MSR device granularity	per CPU/per NUMA node	per CPU
CSR device granularity	per NUMA node	per CPU

To make the new device driver extendable, I decided to use a template based design. There are two different templates possible: processor definitions and parameter definitions. Processor definitions describe a certain processor type by their vendor, family, stepping, and enabled features. Parameter definitions define the register type (e.g., MSRs or CSRs), the register index, the bit-mask that defines the related bits within the register, a number of valid settings, and the processors for which this knob is available. Examples for these definitions are given in Listing 6.1 and Listing 6.2. When preparing the device driver for compilation, source code is generated that reflects the definitions. The compiled driver then contains all information available at preparation time. When the driver is loaded, it checks for each knob definition whether it is available on the current system by assessing the assigned processor definition. The available features are then made available via pseudo files in the `/dev/` file system. The complete process of building and loading the kernel module is depicted in Figure 6.2.
The device driver establishes two different folders for the pseudo files, one for knobs that are available per software thread (CPU) and one for processor settings that are available per NUMA node, i.e., per processor. In each of the two folders, there is one definition file that provides information about the available items and multiple files to change these settings based on the given granularity. I demonstrated the usage of this device driver in [SM13].

```
//#vendor
X86_VENDOR_INTEL
//#families
0x06
//#models
0x2A,0x2D
```

Listing 6.1: `x86_adapt` processor definition for Intel Sandy Bridge processors

```
//#description
Disables DCU Prefetcher. 0 == enabled; 1 == disabled
//#device
MSR
//#register_index
0x000001a4
//#bit_mask
0x4
//#processor_groups
nehalem, sandybridge, westmere, ivybridge, haswell, broadwell, skylake
```

Listing 6.2: `x86_adapt` knob definition for DCU Prefetcher dissablement

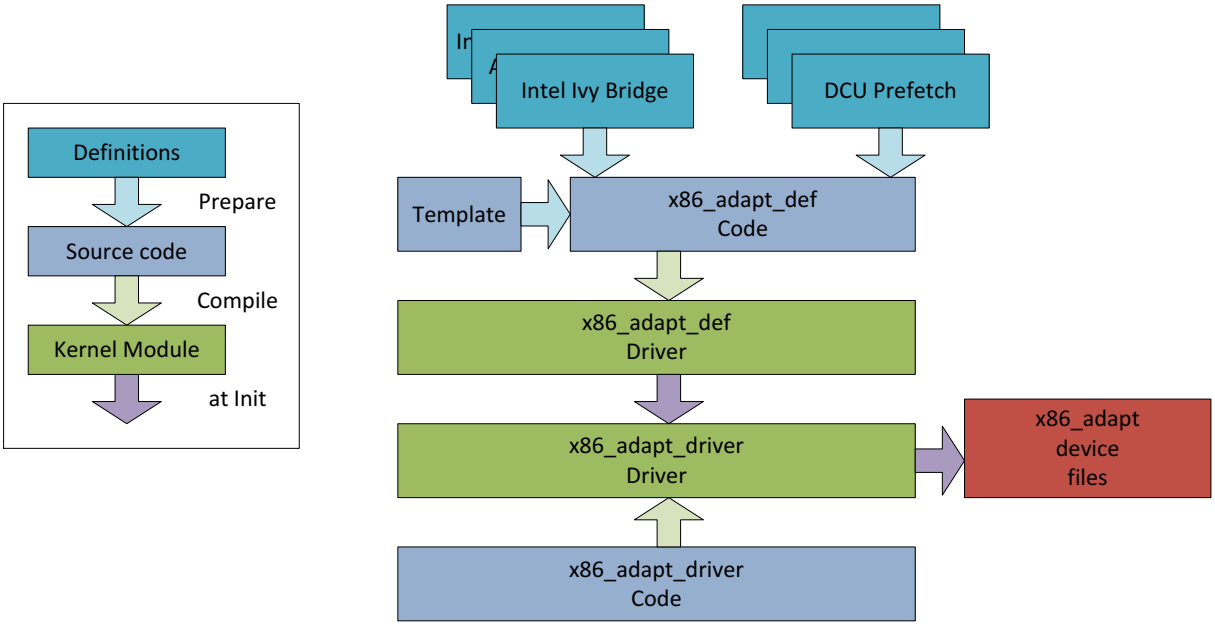


Figure 6.2: `x86_adapt` preparation, compilation and initialization phase

6.2 Energy Efficiency Metrics for VampirTrace and Score-P

Based on the concept presented in Section 5.4.1, I implemented an extension for VampirTrace that is able to handle most different status element collectors. These status element collectors are provided to VampirTrace as shared objects that can be loaded by the dynamic linker at runtime, based on which properties should be recorded. The extension is called "VampirTrace plugin counters" and provides an interface to implement new plugins in the programming language C. I introduced it in [STHI11]. The components of the interface can be distinguished into *metric definitions* and *plugin definitions*. Both are defined in the header file `vt_plugin_cntr.h`. In this section, I provide an overview of the interface and an overhead analysis, discuss changes from the Score-P implementation, and show three examples that demonstrate how energy efficiency information can be captured.

6.2.1 Possible Metric Types

In Section 5.4.1, I defined four criteria that a status element collector interface should be able to describe: *spatial scope*, *value range*, *temporal scope*, and *information type*. The interface supports developers by providing mechanisms to define each of these for the provided metrics. The spatial scope can be defined as:

- per-thread, where each software thread provides its own status elements,
- per-process, where all threads of one process share a property,
- per-host, where the status element describes a property of a compute node, and
- global, where a property is used to describe the status of the whole workload

Since VampirTrace is agnostic to the compute node internal architecture, other spatial scopes like per-core or per-socket are not supported and have to be implemented informally. The respective metrics are counted on the lowest ranked thread. For example, in an MPI parallel program, per-host metrics are counted on all processes that have the lowest MPI rank within their compute node. In a hybrid-parallel program, global metrics are collected by the master thread of MPI rank 0. This can lead to a situation where some threads have to suffer a higher performance perturbation than others if status

elements are collected at every event. To limit the overhead, the interface supports the implementation of asynchronous plugins.
Since properties with a different spatial scope than a single software thread cannot be mapped to a monitored behavior of such a thread, its status elements can be passed asynchronously after the monitoring of the workload ends. In this case, the plugin is marked as *asynchronous post-mortem*.
Alternatively, plugins can provide their asynchronous status elements with each event. In this case, the passed status elements have to be sorted according to their time. Additionally, the "oldest" status element must be measured after the last event and before the current event. Supplementary, the asynchronously measured status elements can be directly written to an internal buffer within VampirTrace. This buffer is read and cleared whenever an event is processed. If the internal buffer is full, further attempts to add status elements fail and return an error code. Such asynchronous plugins are defined as *asynchronous on-event* or *asynchronous callback*. Since the performance perturbation is again not balanced if a coarser spatial granularity is used, these should be used with caution and preferably only for properties assigned to single software threads.
With regard to the information type of a metric, plugins can define whether their metrics are backward looking (`VT_PLUGIN_CNTR_LAST`), instantaneous (`VT_PLUGIN_CNTR_POINT`), or forward looking (`VT_PLUGIN_CNTR_NEXT`). This enables developers to describe transitions, instantaneous measurements, and statistical status elements. In addition, the interface supports accumulating metrics. These are used, for example, in PMUs, which are already supported by VampirTrace via PAPI. Finally, plugins can define the *data type* of the metrics to be `double`, `float`, `uint64_t`, or `int64_t`.

6.2.2 Implementation Details

VampirTrace is configured via environment variables, which are read when the monitoring of the workload is initialized. The front ends define when VampirTrace is initialized. This can be before the main routine is executed or with the occurrence of the first event. Depending on the content of the environment variables, VampirTrace allocates buffers, registers PAPI metrics, and enables profiling, tracing, and other sub-components. Plugin counters are also configured accordingly. To register a plugin and associated metrics, the environment variable `VT_PLUGIN_CNTR_METRIC` has to be set. Within this variable, plugins and metrics are encoded as follows:
`VT_PLUGIN_CNTR_METRIC=<plugin`$_0$`>_<metric`$_0$`>(:<plugin`$_n$`>_<metric`$_m$`>)`*
Each entry encodes a plugin and a metric that is provided by this plugin. Multiple entries can be concatenated. The default separator "`:`" can be overridden. According to this variable, the encoded plugins are loaded via the dynamic linking library `libdl`. If multiple metrics are registered that are provided by a single plugin, this plugin is only loaded once. After the plugin is loaded, VampirTrace captures plugin information like function pointers, the spatial scope, the information type, and the used interface version. Afterwards, it passes a function that provides VampirTrace compatible timestamps to the plugin if the plugin is asynchronous. VampirTrace implements a number of possible status element collectors that provide timestamps. However, only one of these is used for a single measurement. Since these status element collectors are interchangeable, plugins cannot rely on a standardized mechanism to capture timestamps for asynchronous measurements. With the passed function, they are always able to provide correctly formatted timestamps. Afterwards, the plugin is initialized. Here, the plugin is able to check hardware and software dependencies and can return an error code if these are not available. Afterwards, the metric descriptions encoded in the environment variable are passed to the plugin. The plugin can extend each metric description to a number of metric definitions. For example, if the environment variable is set to `foo_*`, the description `*` is passed to the plugin, which can then return descriptions for the metrics *bar* and *baz*. If the spatial scope includes the current process, the metric is registered and enabled for the thread that initializes VampirTrace. For every new thread, the metric is also registered and enabled if the given spatial scope includes the thread. VampirTrace disables the monitoring of threads if they are currently considered to be inactive and re-enables them as soon as they become active again. Such control events are passed to the plugins to reduce the amount of collected status elements. Whenever a front end

observes an event, status elements of asynchronous on-event, asynchronous callback and synchronous metrics are collected. Whenever the monitoring of a thread is terminated (e.g., when the monitoring workload ends), asynchronous post-mortem status elements of associated metrics are collected.

6.2.3 Overhead Analysis

To test the resulting overhead, I use a minimal synchronous and a minimal asynchronous post-mortem plugin on the Intel Xeon E5-2680 v3 test system that is described in Section A.3.3. I use the GNU Compiler Collection (GCC) in version 5.4.0 for VampirTrace, plugins, and the simple test program that is depicted in Listing 6.3. The plugins are described in Listing D.1 and Listing D.2. I vary the number of calls to the function `foo` between 100,000 and 500,000. Thus, between 200,002 and 1,000,002 events are generated from the compiler instrumentation. I use the VampirTrace profiling tool to capture the runtime of the function main. This excludes the time for setting up the measurement environment and the time for postprocessing the collected data. I repeat every measurement ten times and use the median for further calculations. I distinguish between the costs for adding a plugin and the costs for adding a metric. To do so, I register the simplistic synchronous plugin and define one, two, three or four metrics for this plugin. As expected, the overhead introduced by plugin and metrics scales linearly with the number of events, and thus with the number of gathered status elements, depending on the number of registered metrics. This is depicted in Figure 6.3. Here, the single series define the measurement overhead when a specific number of metrics is registered. The slope of these series describes the costs for a single event. I retrieve the costs by using linear regression for $t = \alpha * m + \beta$. The difference between these costs, i.e., the costs for adding a metric is in this case 18.3 $\frac{\text{ns}}{\text{Event}}$, which is at least one order of magnitude lower than the pure reading of traditional performance counters [Wea15]. Since the plugin does not capture any data, the overhead for a real world example would be significantly higher, depending on the complexity of the used software. The assumed costs for a plugin without an active metric are 52.7 $\frac{\text{ns}}{\text{Event}}$, which is 18.3 $\frac{\text{ns}}{\text{Event}}$ lower than the costs for a plugin that provides one metric. 71 % of this time is spent in VampirTrace's internal routines that are not affected by registering plugins. The remaining 15.3 $\frac{\text{ns}}{\text{Event}}$ are the runtime overhead for registering a plugin.

The number of status elements reported by asynchronous post-mortem plugins does not influence the measurement time, even though the overhead for registering a plugin is still measurable for the worst-case example that is used in this section. In real-world examples, the overhead could not be noticed [STHI11]. Still, the runtime of applications is significantly increased if tracing is enabled. This can be explained with two factors. Foremost, VampirTrace post-processes data by re-reading and re-writing the created log files. This process is inherently dependent on the number of written status elements. Additionally, the plugin creates VampirTrace compatible timestamps for each reported value via linear interpolation.

```
void foo()
{
}
void main()
{
  unsigned long long i=0;
  for (i=0;i<NUM_CALLS;i++)
    foo();
}
```

Listing 6.3: Minimal program to determine overhead

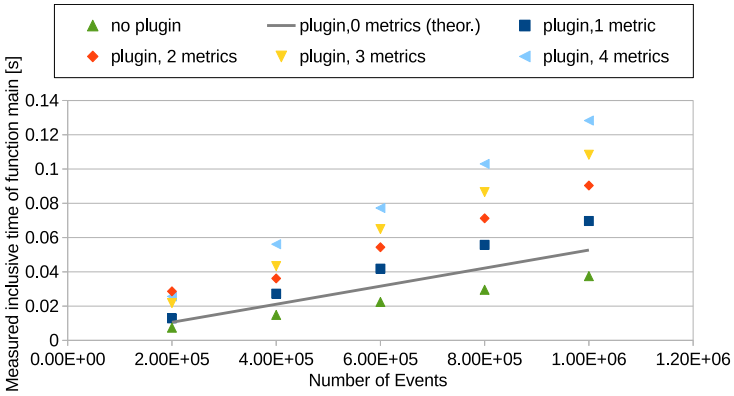


Figure 6.3: Overhead

Figure 6.4: Adding power consumption information to traces via Dataheap and VampirTrace. (OpenMP parallel NAS benchmark BT, Intel Xeon E5-2680 v3 test system, OpenMP instrumentation)

6.2.4 Changes for Score-P

The interface has later been translated to Score-P, where several changes have been applied. The command line interface now has two levels. Plugins are defined in one environment variable, and in other variables depending on the plugin names, the respective metrics are declared. This reduces the possible length of the environment variables. Synchronous metrics can now define whether a new status element is available. The internal handling is now better able to cope with asynchronous plugins, which have now no overhead except for the finalization stage when the results are retrieved. But also, the general overhead has been reduced significantly. More details and an analysis for the expected overhead is described in [STI+17].

6.2.5 Use Cases Related to Energy Efficiency

In [STHI11], I describe various plugins that enable users to measure energy efficiency related metrics. In [KHN12], Kluge et al. describe *Dataheap* – a scalable distributed infrastructure to store and retrieve performance monitoring data. This infrastructure is, for example, used to store power measurement or I/O data as described in [KHN12]. The used plugin is asynchronous post-mortem, which eliminates most of the runtime overhead. Figure 6.4 depicts a resulting trace that is visualized with the performance analysis tool Vampir [BWNH01]. The trace can also be parsed to compute the average power consumption when executing a specific region. Alternatively, power consumption can be measured by using processor internal monitoring capabilities that are available in current processors and discussed in Section 2.4. In [HIS+13], which I co-authored, Hackenberg et al. use a plugin that retrieves RAPL measurements asynchronously. The measurement infrastructure provides a better spatial resolution, e.g., per DRAM channel and per processor. Since VampirTrace and Score-P do not support a description of these scopes, the properties are mapped to an element group that is available – per-host. Figure 6.5 shows the power consumption of processors (packages) and DRAM when executing the OpenMP NAS Parallel Benchmark BT.

In addition to the power consumption information, it is also essential to record properties that influence the power consumption. The activity factor α can be estimated by using PMUs, but also the frequency and the usage of C-states plays an important role. In [STHI11, Section 3.1], I lay out how such a plugin can be implemented via kernel instrumentation. This plugin provides information about frequency and C-state changes by using the perf infrastructure, which provides access to kernel probes. The capturing of transitions at this level provides accurate timing information but also has some downsides. If the event rate is too high, the measurement is significantly influenced. Furthermore, the hardware can override the operating system decisions [Int13b, Section 4.2.4.5]. Alternatively to the kernel instrumentation, statistical status elements can be captured from accumulating registers. These exist for frequencies [Adv13, Section 2.5.6], [Int15a, Section 14.2] and core and package C-states via residency counters [Int15a, Table 35-13]. To match the hardware information to software threads, the plugin constantly monitors the scheduling. I exemplarily visualize the average number of cycles that cores reside in CC6 in Figure 6.6.

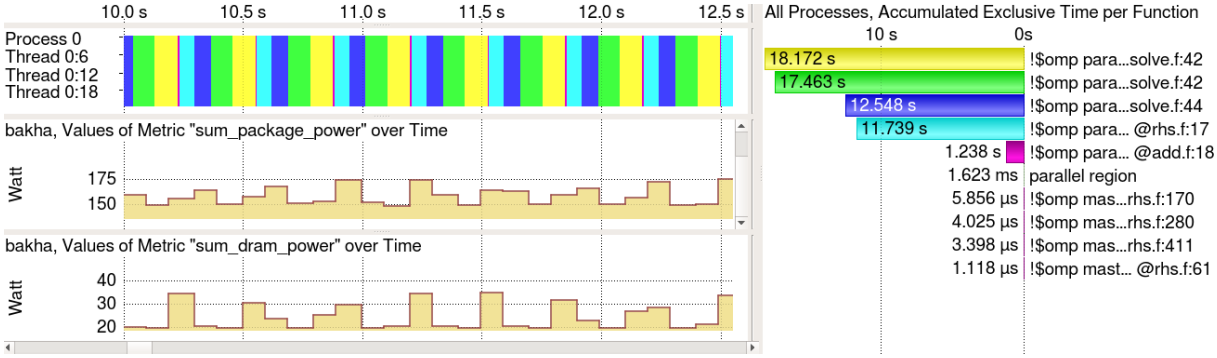


Figure 6.5: Adding power consumption information to traces via x86energy and VampirTrace. (OpenMP parallel NAS benchmark BT, Intel Xeon E5-2680 v3 test system, OpenMP instrumentation)

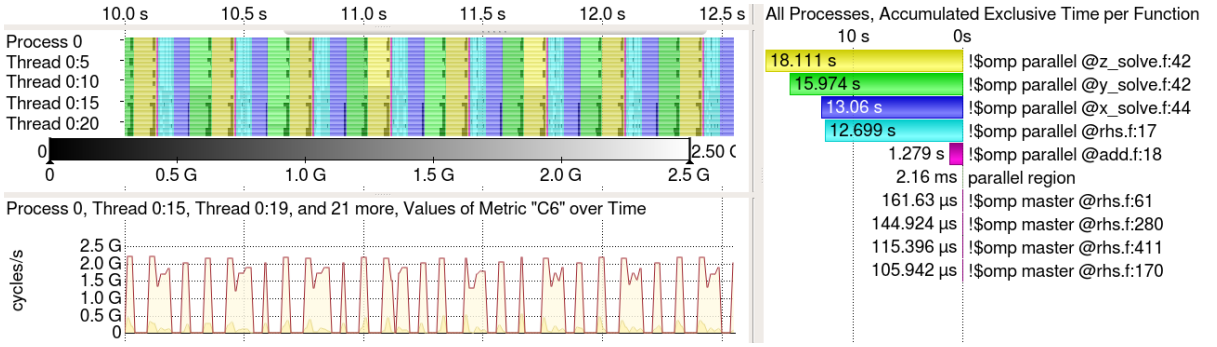
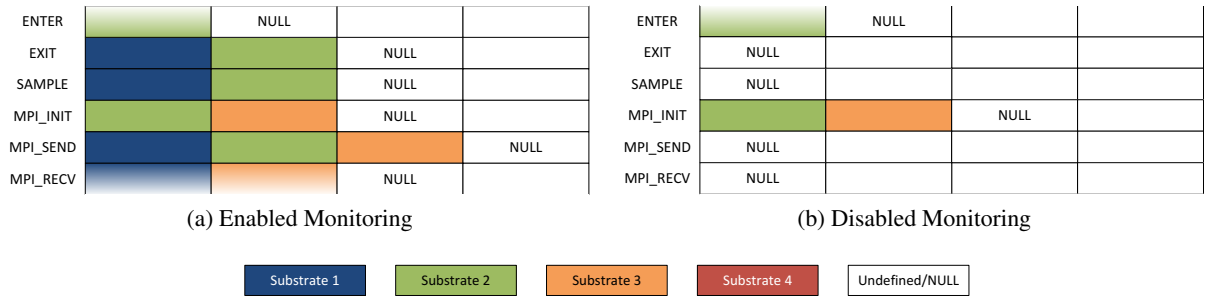


Figure 6.6: Adding C-state information to traces via plugin. Darker regions in the timeline display represent haltet hardware threads. Lighter regions are unhaltet. The average CC6-usage is displayed below.

Other researchers also developed plugins for measuring power consumption. Chasapis et al. describe a plugin for the library PMLib to analyze the energy efficiency of HPC storage systems [CDKL14]. Ilsche et al. describe a plugin that retrieves power monitoring data from a proprietary measurement infrastructure that uses data acquisition cards from National Instruments in [HIS+13]. Knobloch et al. present a solution for node level power monitoring on IBM systems [KFH+14]. Hackenberg et al. implement a solution to measure the power data provided by BULL compute nodes in [HIS+14]. Cray also implements a power measurement solution called pm_lib [FCGS14]. Hart et al. describe a plugin in [HRD+14] where they include power data in application traces and use this plugin, for example, to show aliasing effects of the measurement. Minartz implements a plugin that polls power consumption information from a database for further analysis in [Min13, Section 6.2].



(a) Enabled Monitoring

(b) Disabled Monitoring

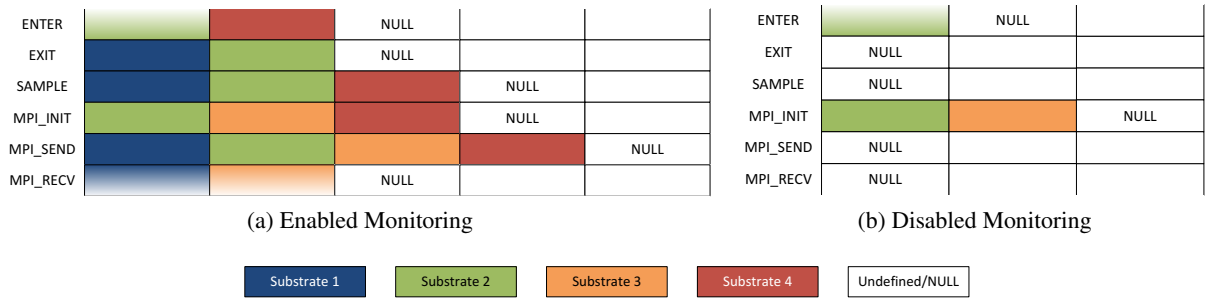
Figure 6.7: Exemplary filling of `scorep_substrates` data structure with functions from different substrates. In this example, only three substrates are supported. Therefore, the fourth substrate cannot be registered.

6.3 Enabling new Back Ends for Measurement and Tuning

In [SM13], I presented an extension to the performance measurement tool VampirTrace which enabled optimizations at region level. In this publication, I introduced the `libadapt` library, which can be used to enable region-based optimizations. However, the integration in VampirTrace provided different short-comings: the development of VampirTrace has been abandoned, the integration of performance measurement in the infrastructure is deep and therefore bears a significant runtime overhead, and the integration of the tuning mechanism is static and cannot be replaced easily. In this section, I describe how I implemented an interface for Score-P that tackles all of these issues. First, I describe implementation details and lay out overhead numbers. Afterwards, I present three different back ends for the interface: one plugin for testing the frequency scaling of regions and two energy efficiency tuning plugins for region-based and balancing-based tuning, respectively. However, to increase the readability and clarify the meaning, I will use the term "substrate" whenever I mean a Score-P back end. I presented interface details to the scientific community in [STI+17].

6.3.1 Existing Back End Implementation in Score-P

At the beginning of my work, Score-P implemented an internal substrate interface. This interface defines a number of events that substrates can use to follow events processed by Score-P. The substrate management system holds an array that stores the function pointers of the different substrates for each of these events. At initialization, the substrate management collects the functions from the existing substrates profiling and tracing. These are then added to the event array, where each row holds a NULL terminated list for the respective event. This is depicted in Figure 6.7. The resulting array is then used by Score-P whenever an event is processed. The event index is used to select the list. Afterwards a function iterates over the registered functions until the NULL pointer is found. The main weakness of this implementation is that the array size, i.e., the number of supported substrates, can neither be extended nor reduced at runtime. To overcome this issue, the array could be implemented as a list of lists. However, this would mean that another pointer-dereferencing would have to be executed at each event, which would increase the runtime overhead. Alternatively, the number of possible substrates could be increased. This would increase the memory footprint even if no additional substrate would be registered. Another issue is the significant runtime overhead that is introduced whenever a user instrumentation en- or disables the recording of events. In this case, the array content is completely re-written. To overcome these obstacles and to enable developers to write own back ends, I changed the internal management and externalized the new interfaces. This is described in the next section.



(a) Enabled Monitoring

(b) Disabled Monitoring

Figure 6.8: Exemplary filling of `scorep_substrates` data structure with functions from different substrates. The array size is determined at initialization time. Since fewer substrates register events for disabled monitoring activity, the respective array is smaller.

6.3.2 Changes to the Existing Back End Implementation

In the initial substrate design, there is no distinction between different types of events. Thus, information that stems from the monitored workload is provided in the same way as information about the internal processing of the integration. The latter cannot be ignored when the monitoring of the workload is disabled and is typically registered to be executed regardless of the monitoring status. Therefore, I split the array into two different arrays. One array holds the *management* functions and is used whenever the integration provides new information about the workload, e.g., whenever a new thread is created. The other array holds *event* functions. Event functions are called whenever an event is provided by the existing front ends. As basic information, each of these calls receives the location (e.g., the thread) that provides the event and the current time stamp. To avoid the overhead when en- and disabling the measurement, there are two copies of the event array. If the user instrumentation switches between active and inactive monitoring, a pointer is switched to refer to the respective array. All arrays are initialized at the beginning of the monitoring.

To enable a varying number of substrates, the arrays are initially allocated large enough to hold event functions for all registered substrates. After management and event functions of all substrates are registered, these lists are packed. Afterwards, the offset represents the number of substrates that registered for the event with the highest number of substrates plus one for the NULL terminator. For example, in Figure 6.8a, four substrates registered for event MPI_SEND. Therefore, the offset is 5. Only two substrates registered to be called for event MPI_INIT when monitoring is disabled. The resulting offset is 3. After the offset is determined, it is padded to limit the number of used cache lines per event. Thus, it is rounded up to 1, 2, 4, or $\frac{\text{cache line size}}{\text{function pointer size} * 2}$. To process the functions for a specific event, the NULL terminated sublist at `offset*event-nr` is accessed.

6.3.3 Substrate Plugin Interface

To enable developers to implement own substrates, parts of the internal interface are exported. While the event functions for plugins are the same internally as externally, management functions are masked behind a struct that is also used to define the plugin. In this section, I describe the interaction between Score-P and a substrate plugin focusing on the handling of management functions.

When Score-P is started, it reads the environment variable `SCOREP_SUBSTRATE_PLUGINS`, which defines the plugins that shall be loaded. If, for example, this environment variable is set to foo, Score-P will attempt to load the shared object `libscorep_substrate_foo.so` and try to load the plugin definition from this plugin. If this succeeds, Score-P initializes the plugin. Afterwards, it passes *callback* functions to the substrate library. These callbacks can be used to access Score-P internal functionality. For example, they can be used to determine the result folder where traces and profiles are stored. Plugins can use this information to store own data within this folder. Other callbacks include the inter-process-communication (IPC) layer, functions to access thread-local data, and functions to gather meta data

from Score-P internal definitions. The IPC layer enables plugins to share information between multiple processes that do not share a common memory segment. This functionality can be used after a multi process paradigm like SHMEM or MPI is initialized, which is also passed to a plugin. Afterwards, Score-P reads the event functions for enabled and disabled monitoring. After this point in time, Score-P can create new definitions that can be referred to in upcoming event functions, e.g., definitions of instrumented functions or measured metrics. Later, Score-P assigns an identifier to the plugin, which it can use to read and write thread local data. Afterwards, the plugin will be informed whenever a new thread is created or deleted. Between these management events, monitoring information is passed to the event functions of the plugin. When the monitoring is finalized, Score-P calls the plugin so that it can flush information or free resources. All calls that I mentioned in this section, except for the plugin definition, are optional. Thus, plugin developers can decide themselves which functionality they need to achieve their target.

6.3.4 Overhead

To examine the overhead of the interface, I conduct an experiment that is similar to the test that I presented in Section 6.2.3 using the same system and test program (Listing 6.3). The runtime is measured by using Score-P's profiling substrate. To extract the runtime from the resulting profile, I use the `cube_stat` tool that lists the inclusive runtime of the main function. Instead of metrics, I register a minimal substrate plugin that implements functions for enter and exit events as defined in Listing 6.4[1]. Again, I use different number of calls to the instrumented function `foo`, repeat the measurement of each problem size ten times and use the median result. The resulting runtimes are depicted in Figure 6.9. The measured values are points and the lines represent the linear regression of these points. The difference of the slopes of the two linear fits represents the costs for a single call to the substrate, which happens to be 3 ns (12 cycles).

6.3.5 Use Cases Related to Energy Efficiency Tuning

In this section, I present three different use cases targeted at energy efficiency tuning purposes. Alternatively, the interface also enables developers to implement other back ends with purposes beyond energy efficiency. Trommler created an extension that filters short-running functions to lower the performance perturbation of the measurement[2]. In [STI+17], I present a plugin that writes EFGs as graphviz files. Figure B.1 and Figure B.2 have been created with this substrate.

```
static void enter_region(
 SCOREP_Location*     location,
 uint64_t             timestamp,
 SCOREP_RegionHandle  regionHandle,
 uint64_t*            metricValues ){
}
static void exit_region(
SCOREP_Location*     location,
uint64_t             timestamp,
SCOREP_RegionHandle  regionHandle,
uint64_t*            metricValues ){
}

/* Register event functions */
static uint32_t get_event_functions(
 SCOREP_Substrates_Mode         mode,
 SCOREP_Substrates_Callback** returned)
{
 functions=calloc(...);
 functions[SCOREP_EVENT_ENTER_REGION]=enter_region;
 functions[SCOREP_EVENT_EXIT_REGION]=exit_region;
 *returned = functions;
 return SCOREP_SUBSTRATES_NUM_EVENTS;
}
```

Listing 6.4: minimal substrate plugin (excerpt)

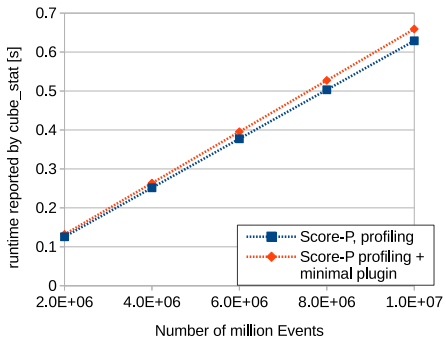


Figure 6.9: Measured overhead for a minimal substrate plugin that registers for enter and exit events

[1] The complete source is given in Listing D.3

[2] `https://github.com/Ferruck/scorep_substrates_dynamic_filtering`

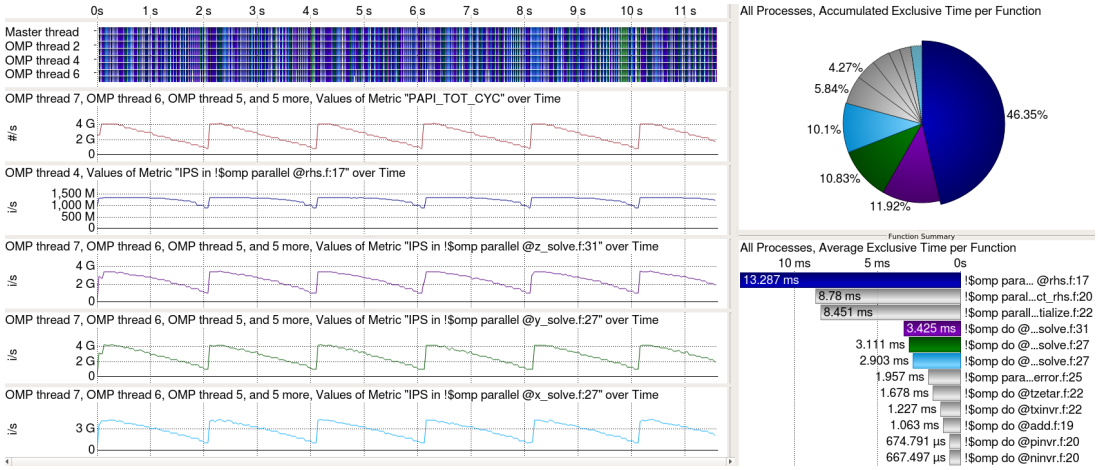


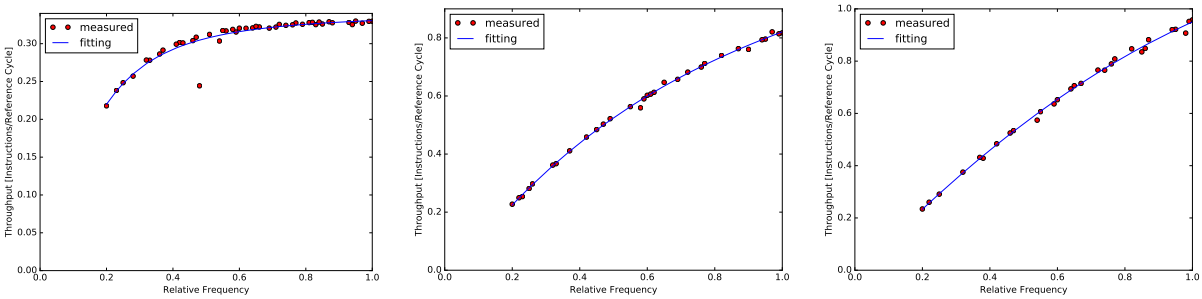
Figure 6.10: Vampir displaying measurements of NAS Parallel Benchmark SP (OpenMP, size A) on the Intel Core i7-6700K test system with the frequency changing every 1 second (red metric). The parallel region `!$omp parallel@rhs.f:17` (marked blue) has a share of 46.35 % on the execution time. It is almost not influenced by the applied frequency – its instructions per second (IPS) are not affected. The next three regions (`!$omp do@[x|y|z]_solve`, marked purple, green, and cyan) have a total share of 32.85 % and scale linearly with the applied frequency.

6.3.5.1 Scaling of Program Regions

The first plugin I describe is rather simple and only registers for the initialization and finalization management event. When the plugin is initialized, it sets up a timer and applies the fastest processor frequency. When the timer expires, the plugin reduces the frequency by a step and resets the timer. At finalization, the timer is canceled. While such a plugin does not perform any useful work by its own, it can be used in combination with tracing to determine energy-efficient frequencies that should be used by the monitored software. According to the model that I introduced in Chapter 3, the energy efficiency of a region increases if the throughput remains constant and the power consumption is reduced. Based on Equation 2.1, the power consumption of integrated circuits decreases when the applied frequency is reduced. Thus, it is safe to assume that the energy efficiency of a region is increased if its throughput is equal to the default throughput and the frequency is below the default frequency. Power metrics are not needed to generate this information. Hence, this approach can be used on all systems that support frequency scaling and the recording of hardware PMCs.
In Figure 6.10, I show that while the majority of the functions of the selected workload scales with the applied frequency, the region with the highest execution time `!$omp parallel@rhs.f:17` is almost not influenced. The next three functions according to the runtime scale linearly with the frequency. This information can be used in upcoming executions of the workload if another plugin uses the passed status elements to distinguish the different functions and apply the respective most suitable processor frequency.

6.3.5.2 Region-based Energy Efficiency Tuning

In [SM13], I show that it is possible to use a common instrumentation for both: measuring performance data and optimizing the energy efficiency. In this work, I use Performance Monitoring Counters as an indicator to detect memory boundedness and predict efficient frequency configurations. In [MSHN17], which I co-authored, Molka et al. describe a method for finding more suitable PMCs for detecting memory-boundedness and apply it to an Intel Haswell-EP processor. However, other information sources can also be used to generate region-based optimizations. In this section, I describe a plugin that uses the optimization library *libadapt*, which I initially described in [SM13]. This library uses configuration files,



(a) `!$omp parallel@rhs.f:17` (b) `!$omp do@z_solve.f:31` (c) `!$omp do@y_solve.f:27`

Figure 6.11: Scaling of single regions with the applied frequency. The data that is used is gathered from the trace depicted in Figure 6.10. The used fitting function is listed in Equation 6.1 and has been optimized with scipy's least squares method.

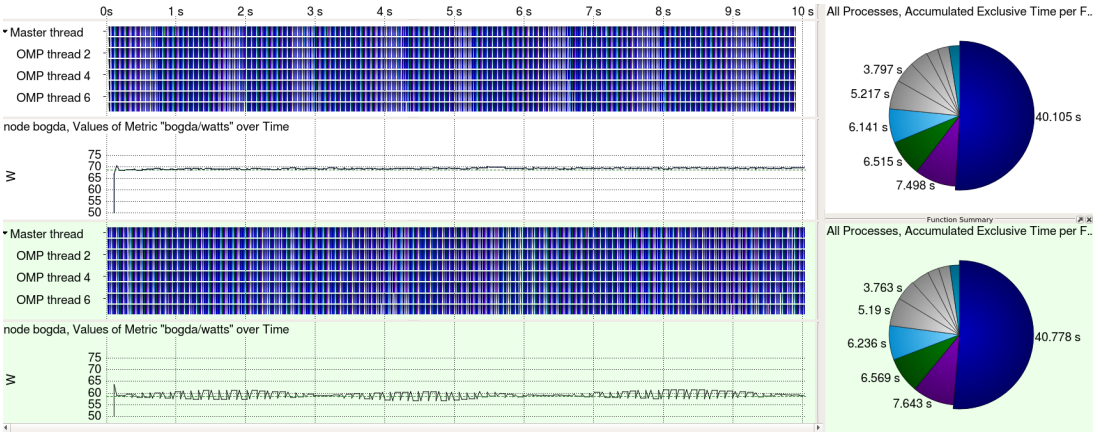


Figure 6.12: Example of `libadapt` plugin, use of the configuration file that is created from the trace depicted in Figure 6.10. The default execution time is 9.9 s, the average power consumption is 68.7 W (upper trace). The optimized configurations execution time is 10.0 s with an average power consumption of 58.7 W (bottom trace).

in which users can describe configurations that should be applied when certain events occur. The configuration descriptions include support for processor frequencies, thread concurrency, low-level hardware configurations via `x86_adapt` (see Section 6.1), and idle state limitations. In the previous section, I presented a plugin that can be used to determine efficient frequencies for single functions. The resulting trace is parsed to create a profile for the single functions that describes how their throughput scales with the applied frequency. Profiles of the three functions with the highest share of the runtime are depicted in Figure 6.11. It calculates three parameters α, β, and γ using the least squares method on Equation 6.1 with f_r being the relative frequency and IPS the measured throughput.

$$IPS(f_r) = \alpha(\pi/2 - \arctan(\beta f_r + \gamma) * f_r) \tag{6.1}$$

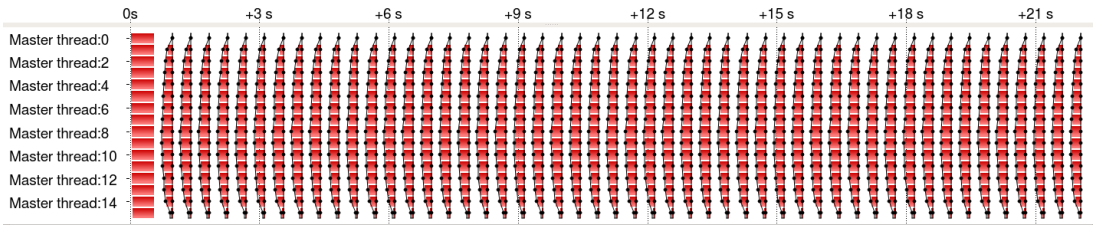
This equation has been chosen to be able to describe the compute bound part, where the fraction $\arctan(\beta x + \gamma)$ is close to zero and there is a linear correlation between frequency and throughput. If f_r gets large enough that the workload is memory-bound, the $\arctan(\beta f_r + \gamma)$ part becomes close to $\frac{\pi}{2}$. Thus, the IPC is constant. The script then uses the derivation $\frac{d}{df_r}[IPS(f_r)]$ to retrieve the relative frequency where the slope becomes lower than a certain threshold. The same script then creates a configuration file that libadapt uses to reduce the processor core frequency in the `!$omp parallel@rhs.f:17` region. The resulting trace of the same benchmark with the given configuration is depicted in Figure 6.12.

6.3.5.3 Balancing-based Energy Efficiency Tuning

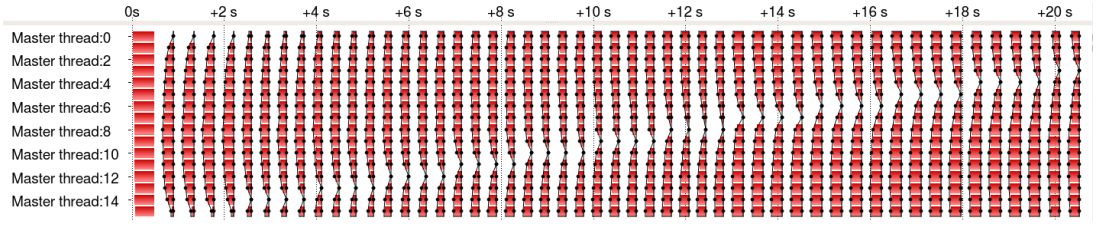
An alternative to region-based tuning is balancing-based tuning, as I described in Section 2.7.2. In parallel simulations, different amounts of work can be attributed to the participating threads and processes. This is depicted for a test workload in Figure 6.13. In this workload, some ranks (initially those around $\frac{size}{2}$) are assigned less work. Thus, they arrive too early in a barrier. The time that is spent in synchronization due to such an imbalance and a following early arrival is called *slack*.
The proposed infrastructure also supports such an approach to increase the energy efficiency of parallel programs. To demonstrate the applicability, I implement an algorithm that is inspired by the one described by Rountree et al. [RLdS+09]. The algorithm assumes that there is a sequence of regions where a computing region is followed by a synchronization region and vice versa. The algorithm assigns regions to functions by gathering an identifier of the current stack whenever a synchronization function is called. Internally, it saves the slack for each function that is to be expected in the following compute region. If the slack is too high, the algorithm applies a frequency that slows down the computation to arrive just in time. I modified the algorithm in the following way:

- The plugin also supports OpenMP parallel balancing-based tuning, which has been proposed by Wang et al. [WSM15]
- To overcome the limited granularity of frequency domains, the plugin supports T-states in addition to P-states.
- The algorithm constantly re-evaluates its decision and increases the effective frequency of the used processor core if the slack tends to be zero and it can be assumed that the current thread is the critical path.
- To avoid a highly frequent changing of frequencies, the plugin smoothens the proposed frequencies and applies the maximum of the last four frequencies that have been selected for the upcoming function.
- If the total time of a compute region and synchronization region pair is too low, no frequency changes will be applied based on its imbalance.

The strategy of the plugin is illustrated in Algorithm C.3. Whenever an enter or exit event is triggered and the respective function is a synchronizing function (e.g., `MPI_Barrier`), the plugin executes the

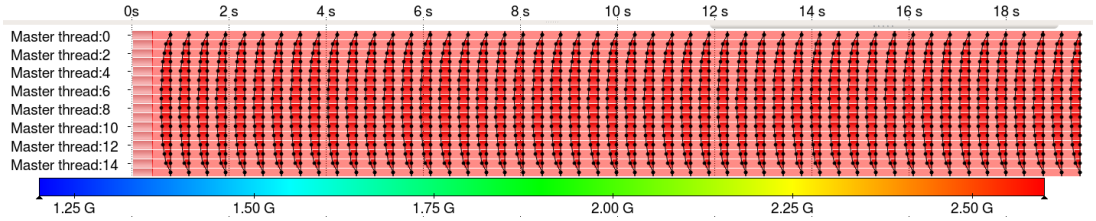


(a) no shift in imbalance

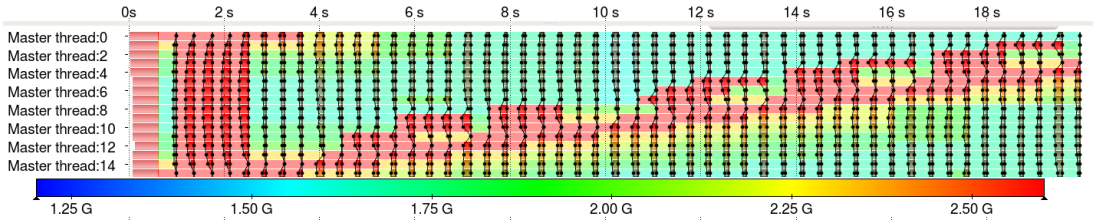


(b) shift in imbalance

Figure 6.13: Test workload (MPI version) without load balancing executed on Intel Xeon E5-2670. The red areas represent times when MPI ranks wait in synchronization.



(a) No shift in imbalance, P-state optimization



(b) Shift in imbalance, T-state optimization

Figure 6.14: Test workload (MPI version) with enabled load balancing executed on Intel Xeon E5-2670. In addition to the synchronization phases, an overlay of the recorded processor frequency is displayed in the trace. Due to the limited granularity of P-states on Intel Sandy Bridge processors, the highest frequency of eight ranks is applied.

respective functions. When the synchronization ends, a new measurement for the previous function (a pair of compute and synchronization regions) is created. Afterwards, a tuned target frequency of the expected follow-up function is applied. A resulting trace can be seen in Figure 6.14a. Here, 16 MPI ranks execute the workload on the Intel Xeon E5-2670 system. While the first eight ranks are executed on the cores of the first processor, the remaining ranks are executed on the second processor. Thus, the critical path is executed on each processor. Since the scope of P-states is per-processor, none of the participating processors is able to lower its frequency. In Figure 6.14b, T-states are applied instead of P-states. Thus, the scope is fine-granular enough to reflect the single ranks. Since the imbalance changes over time, the effective frequency of the cores also changes. Due to the lower effective frequencies, the critical path differs slightly from the critical path that is depicted in Figure 6.13b. Thus, multiple ranks use the highest available frequency. The shift of imbalance also influences the efficiency of the algorithm since the tuning decisions are based on local information.

6.4 Conclusion

In this chapter, I used the concept that I described in Chapter 5 and introduced two interfaces that extend existing performance measurement infrastructures. Prior to this, I presented a kernel module that provides access to different processor registers, for example, to control power-saving mechanisms or read information about their usage. However, the first interface enables programmers to augment the existing performance data with supplemental information like power measurements. Based on this, scientists are now able to analyze the energy efficiency of applications. I described the arising overhead for the VampirTrace implementation and listed changes for the Score-P implementation. Furthermore, I showed three different plugins that collect energy-related information from different data sources. The described interface is actively used by the community, as I demonstrated with a number of references. Plugins that provide power and energy information can be used to evaluate the energy efficiency of programs and single regions as Hackenberg et al. have shown in [HIS+14]. They can also be used to find issues of the power measurement infrastructure [HRD+14].

The second interface allows developers to use the measurement infrastructure for different purposes in addition to the traditional targets of writing timelines and profiles. I present three use cases targeted at analyzing and tuning the energy efficiency of applications. In the following chapter, I will apply these plugins to HPC applications and show that the implemented concept is applicable to real-world scenarios.

7 Energy Efficiency Tuning Evaluation

"We're improving!", Elan the Bard
The Order of the Stick – Blood Runs in the Family by Rich Burlew

In Chapter 5, I described how an infrastructure for a combined measurement and tuning platform should be designed. I detailed what kind of information has to be supplied to different back ends. I described how I implemented an interface to supply additional metrics that characterize the current system status and showed how these can be used to analyze the energy efficiency of a program in Section 6.2. I presented an interface to implement different back ends for such a combined infrastructure in Section 6.3. Furthermore, I showed that it is possible to implement back ends for a region-based offline tuning in Section 6.3.5.2 and a balance-based online tuning in Section 6.3.5.3.
In this Chapter, I use the infrastructure that I specified in Chapter 6 to demonstrate the applicability of the proposed tuning plugins. I describe the used test system in Section 7.1. Results for a region-based offline tuning are presented in Section 7.2, results for a balance-based online tuning are given in Section 7.3.

7.1 Used Test System

The High Performance Computing and Storage Complex II (HRSK-II) located at Technische Universität Dresden's Lehmann Center (LZR) targets energy efficiency at different levels. This includes the re-usage of the system's power dissipation to heat the surrounding buildings as well as a liquid cooling concept by the vendor Bull. Furthermore, the staff decided to allow users of the system to access power saving mechanisms to use the hardware more efficiently[1]. The Haswell based partition of the system debuted in the 45th Top500 list at rank 66 and achieved the 96th rank in the Green500 list. In the most current list (November 2016), the system ranked 107th and 148th with respect to throughput and energy efficiency, respectively. Node power consumption of the Haswell nodes is continuously monitored via the HDEEM and Dataheap infrastructures, which have been discussed in Section 2.4. I give an overview on the used hardware and software in Table 7.1. The test system that is described in Appendix A.3.3 uses the same processor. Thus, more details on the processor architecture can be found there. Simultaneous Multithreading is de-activated. To avoid the re-scheduling of processes, the used batch system Simple Linux Utility for Resource Management (SLURM) pins processes to a set of hardware threads, depending on the number of requested threads. I further pin OpenMP threads by setting the environment variable `GOMP_CPU_AFFINITY`. I query the energy of an executed job with the SLURM accounting tool `sacct`. I prevent influences of processor power variation by running all benchmarks that are compared to each other in a single SLURM job, and consequently on the same nodes. I also run every configuration three times and discuss the result with the median runtime for the discussion.

Table 7.1: Description of test system hardware and used software

Processor	2x Intel Xeon 2680 v3 per compute node
Network	Mellanox Technologies MT27500, 2x Intel Corporation I350 Gigabit Ethernet
Linux Kernel	2.6.32-642.11.1.el6.Bull.106.x86_64
Compiler Suite	Intel Compiler Suite 2016.1.150
Score-P version	Test branch TRY_TUD_substrate_plugins, SVN rev. 11531
MPI version	bullxmpi 1.2.8.4
PAPI version	5.4.3
Power monitoring	HDEEM

[1] https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/X86Adapt

7.2 Region-based Tuning

In this section, I demonstrate the effectiveness of the region-based energy efficiency substrate plugin that I presented in Section 6.3.5.2. To collect optimal target configurations, I use the substrate plugin that I introduced in Section 6.3.5.1 and an analogous plugin for items that are provided by the x86_adapt kernel module that I presented in Section 6.1. I use the latter for testing the uncore frequency, since other parameters did not improve the performance or power consumption.

7.2.1 BT

The benchmark Block Tri-diagonal (BT) is part of the NAS Parallel Benchmarks (NPBs) [BBB+94]. It is written in Fortran and solves "discretized versions of the unsteady, compressible Navier-Stokes equations in three spatial dimensions" according to [dWJ03]. In this section, I use the OpenMP parallel version of the benchmark. I visualize the trace and the runtime profile in Figure 7.1. The main phase, which reflects the vast majority of the total runtime, executes a loop of five OpenMP parallel functions: rhs, x_solve, y_solve, z_solve, and add. The loop is depicted in Figure 7.2, which has been created by the event flow graph substrate, which I mentioned in Section 6.3 and published in [STI+17]. I use two different substrates to test whether power, runtime, and energy consumption of the single regions scale with the applied core and uncore frequency. Their operating principle is laid out in Section 6.3.5.1. I visualize the results in Figure 7.3 and Figure 7.4.

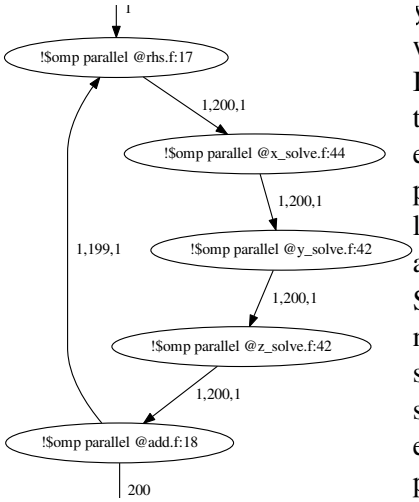
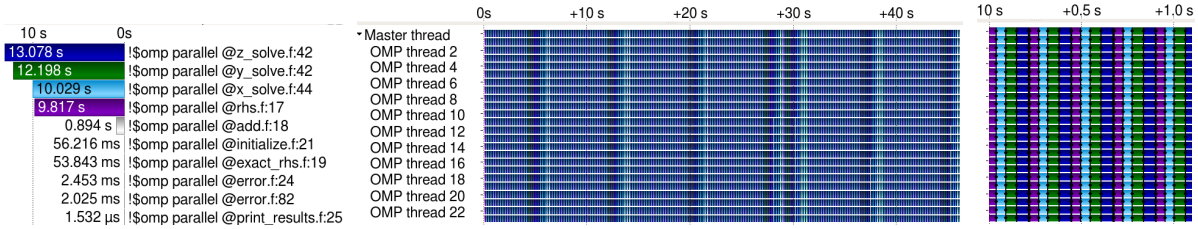


Figure 7.2: Leaf-node-EFG of BT benchmark, main phase

Since an OpenMP parallel region includes a certain synchronization time, the number of executed instructions is not constant and the second part of Equation 3.1, which bases on instruction, throughput, and power readings cannot be used. However, the first part of the equation only uses the runtimes and power consumptions of the regions. It states that the energy efficiency of a configuration c is increased in comparison to the default configuration c_0 if the runtime remains constant but the power consumption is reduced. The individual plots show how the runtime, power consumption, and energy efficiency of selected parallel regions scale with the core and uncore frequency. Core frequency, uncore frequency, runtime, and power are normalized to the reference core frequency, the highest uncore frequency, and the runtime and power at the highest core/uncore frequency, respectively. The five functions of the main loop can be divided into three different categories. The first category includes x_solve. This parallel region benefits from a higher core frequency, but the uncore frequency does not influence its runtime significantly. The second cate-

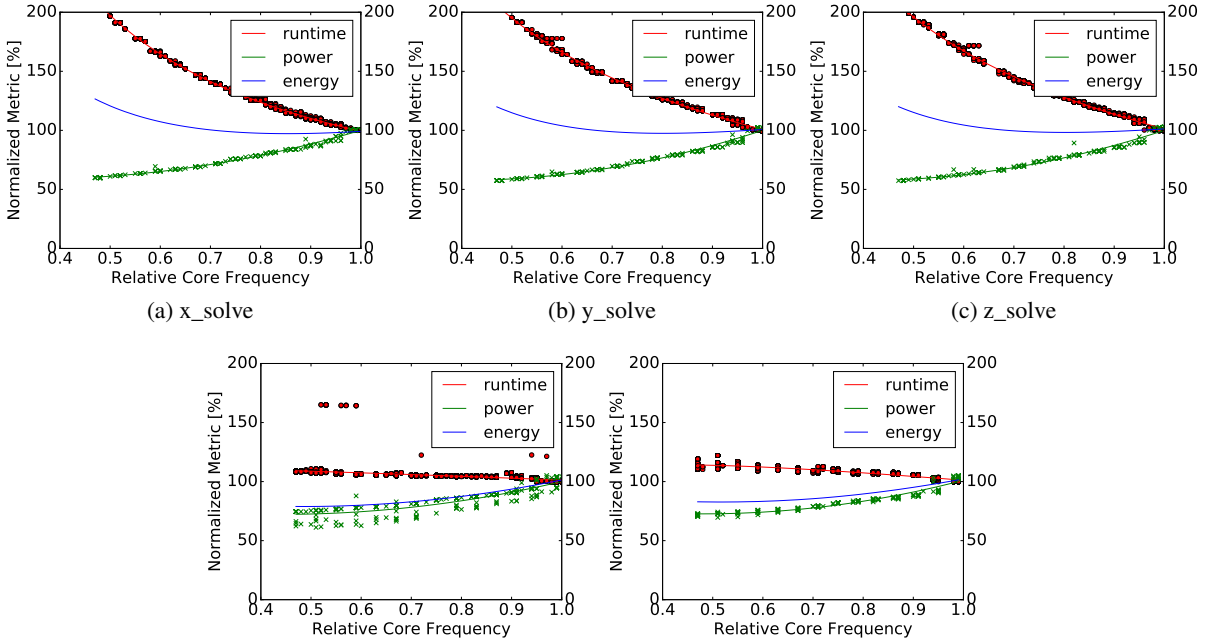


(a) BT benchmark profile, inclusive time of parallel regions for master thread

(b) BT benchmark trace (zoomed out)

(c) BT benchmark main phase (zoomed in)

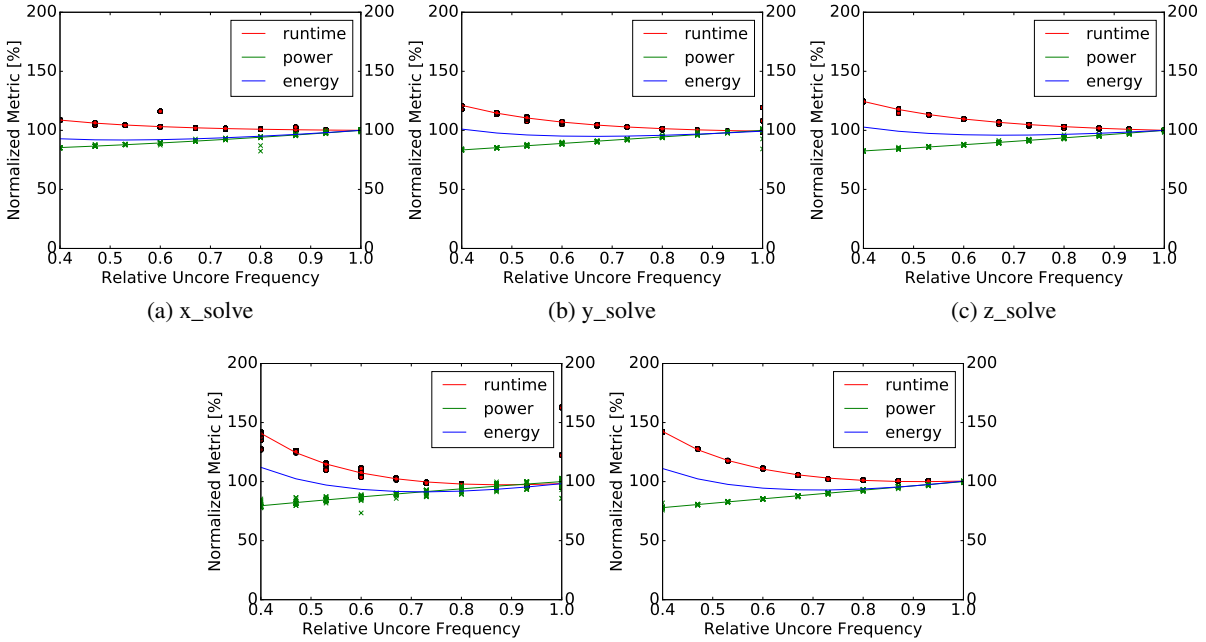
Figure 7.1: Profile and traces for BT benchmark



(a) x_solve (b) y_solve (c) z_solve

(d) rhs (e) add

Figure 7.3: Scaling of runtime, power, and energy with core frequency for parallel regions of BT's main phase. Points mark measured regions, lines show fitted functions. Frequency has been measured with the PAPI_TOT_CYC metric. Power has been measured with HDEEM. The power consumption of a region is the average power consumption of all sampled power measurements. Energy is derived from power and runtime fitting.



(a) x_solve (b) y_solve (c) z_solve

(d) rhs (e) add

Figure 7.4: Scaling of runtime, power, and energy with uncore frequency for parallel regions of BT's main phase. Points mark measured regions, lines show fitted functions. Uncore frequency relates to the applied common minimal and maximal uncore frequency. Power has been measured with HDEEM. The power consumption of a region is the average power consumption of all sampled power measurements. Energy is derived from power and runtime fitting.

```
1  name = ".*bt.*x";
2  function_0: {
3   name="!$omp parallel @rhs.f:17";
4   dvfs_freq_after=2500000;
5  };
6  function_1: {
7   name="!$omp parallel @x_solve.f:44";
8   x86_adapt_Intel_UNCORE_MIN_RATIO_all_before=14;
9   x86_adapt_Intel_UNCORE_MAX_RATIO_all_before=14;
10  x86_adapt_Intel_UNCORE_MAX_RATIO_all_after=28;
11  x86_adapt_Intel_UNCORE_MIN_RATIO_all_after=28;
12 };
13 function_2: {
14  name="!$omp parallel @add.f:18";
15  dvfs_freq_before=1400000;
16 };
```

Listing 7.1: Configuration file for optimizing BT

gory comprises y_solve and z_solve. Like x_solve, the runtime and energy efficiency of these functions benefits from a higher core frequency. However, the duration of these regions is prolonged by 20 % if the uncore frequency is lowered. The third category includes the remaining parallel regions: add and rhs. The runtime of both is mostly independent of the core frequency. A reduced uncore frequency can increase the runtime by more than 40 %.

To calculate the relative energy, I apply the least squares method of Python's scipy package to find a fit for the runtimes ($t(r,f) = \alpha + \beta f + \gamma/f$) and power consumption measurements (polynomial regression, 2nd degree). The projected energy at a specific core and uncore frequency is then calculated by using the fitted functions.

Based on the projected energy and runtime, more suitable settings can be applied to the workload. To do so, I use the substrate plugin that I introduced in Section 6.3.5.2 and apply the configuration file that is shown in Listing 7.1. Based on this file, the adapt library will change the configuration at four points. First, the core frequency is increased to 2.5 GHz whenever rhs is left (lines 3–5). While this might not influence the configuration, when rhs is executed for the first time, it will be needed later. After rhs, x_solve is executed, which changes the uncore configuration when it is entered and exited (lines 6–12)[2]. x_solve is followed by y_solve and z_solve, which share a common configuration of a 2.5 GHz core frequency and 2.8 GHz uncore frequency. The last function in the main loop is add. Since it shares a common efficiency pattern with its most probable successor rhs, it lowers the core frequency for both parallel regions. The total compute cycle as depicted in Figure 7.2 takes approx. 235 ms on average. This means that even an overhead of 1 ms for switching the configuration and the successive delay increase the runtime by less then two percent.

I apply the created configuration file to two different input-sets of the benchmark. All NAS Parallel Benchmarks come with predefined problem sizes[3], which can be used to compare HPC systems of different sizes. In this section, I measure runtime and energy consumption of size C and D, where the size of the used 3-d data structure is 162^3 and 408^3, respectively. Thus, the memory footprint differs by a factor of approx. 16. This can change the characteristics of single regions, since larger datasets are attributed to the individual cores. I present the runtime (as reported by the benchmark) and energy consumption results of the tuning in Table 7.2.

The first result is apparent: the overhead of the general infrastructure is negative. Thus, the runtime of the instrumented application is lower than the one of the uninstrumented version. This pattern is replicable, improves the performance by less than 1 %, and might be caused by cache effects that are introduced by the measurement environment. However, applying the energy efficient configuration does indeed increase the runtime by 1.2 seconds (2.4 %). This can be attributed to three factors. First, the tuning library `libadapt` adds a certain runtime overhead $t(switch, c)$. Second, the target of the tuning had

[2] Two settings define the lowest and highest allowed frequency that the processor should apply to the uncore (in 100 MHz).

[3] listed at `https://www.nas.nasa.gov/publications/npb_problem_sizes.html`

Table 7.2: Runtime and energy consumption of BT benchmark

Problem Size	Instrumentation	Runtime [s]			Energy [kJ]		
		min	**median**	max	min	**median**	max
C	uninstrumented	45.76	**45.63**	46.16	13.87	**13.72**	13.99
	instrumented, default	45.37	**45.57**	45.84	13.91	**13.96**	13.9
	instrumented, tuned	46.33	**46.79**	47.49	12.76	**12.85**	13.03
D	uninstrumented	1008.3	**1008.4**	1008.4	333.7	**333.3**	331.5
	instrumented, default	1004.6	**1004.6**	1005.4	333.3	**333.4**	333.8
	instrumented, tuned	1054.3	**1054.6**	1054.8	314.1	**314.1**	314.4

been energy consumption. Thus, a performance loss is accepted if the energy consumption still decreases. Third, the configuration of the hardware is only applied after a certain delay d of approximately 270 µs as I point out in Section 4.3.3. There are two phases where the applied configuration is probably not optimal in terms of performance: after x_solve and rhs is executed, the uncore and core performance is still reduced for a certain amount of time. However, the maximal loss for these phases is approximately 60 ms if the follow-up regions do scale linearly with both frequencies[4]. The energy efficiency in terms of ETS increases by 7.6 %, which is close to the optimal configuration with no switching overhead and latencies considered. Figure 7.5 depicts the power consumption profile of a single loop for the default and the tuned configuration. Apparently, there are four phases within the rhs region that are indicated by the four different power plateaus. In the default case, all of these phases inflict a power consumption of more than 300 Watts. The DVFS tuning configuration reduces this to less than 270 Watts. The follow-up region x_solve's power consumption is also reduced significantly. At the end of this and the following two regions, the power consumption drops for a short time period. Here, the threads synchronize, which apparently reduces the computing intensity. The add function also reduces its power consumption from 300 to approx. 260 Watts due to the DVFS tuning.
If the configuration is applied at problem size D, the runtime increases by 50 seconds (4.6 %) even though the number of configuration changes remains constant. Therefore, neither the switching overhead $t(switch, c)$ nor the delay d can be responsible for this effect. The increased problem size changes the characteristics of the executed regions, which presumably become more memory-bound. When removing the uncore frequency scaling for x_solve from the configuration, the runtime becomes 1012 seconds, which relates to an increase of 0.3 %. However, the energy consumption for applying the tuned configuration is reduced by 5.7 %. If the uncore frequency tuning is omitted, the energy efficiency gain is only 4 %.

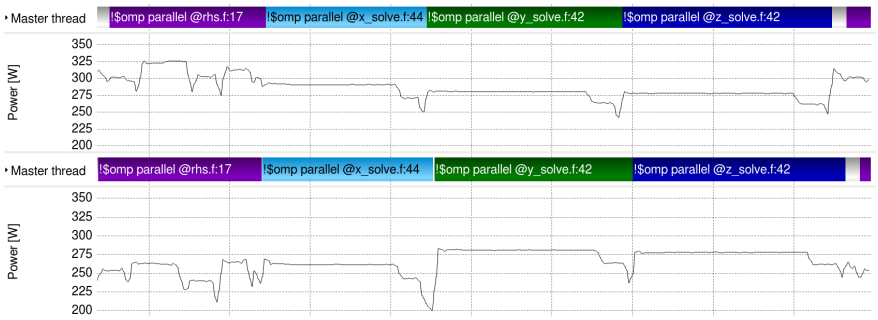


Figure 7.5: Comparison of the BT benchmarks power consumption for default (top) and tuned (bottom) configuration.

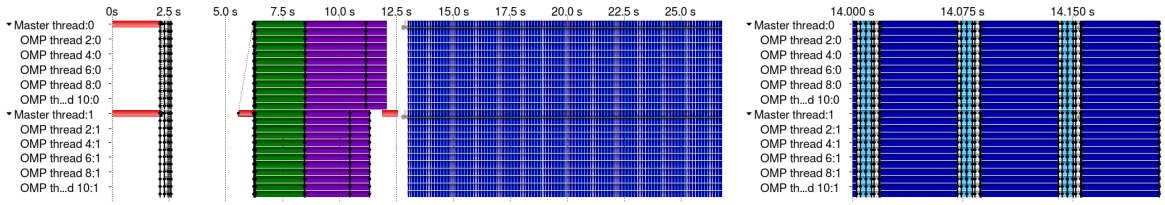
[4]The 60 ms are calculated using Equation 3.4: There are two scenarios, where the configuration $c_\prec$ is not runtime optimal. After rhs is executed, the core frequency is reduced by 52 % ($1 - {}^{1.2\text{GHZ}}/_{2.5\text{GHZ}}$). After x_solve is executed the uncore frequency is reduced by 60 % ($1 - {}^{1.2\text{GHZ}}/_{3\text{GHZ}}$). Each of these phases has a duration d of approx. 270 µs and there are 200 loops that are executed. Thus, the total worst-case runtime loss is $200 * (.52 * 0.27ms + .6 * 0.27ms)$

7.2.2 MiniFE

According to the documentation [HDC+09], MiniFE

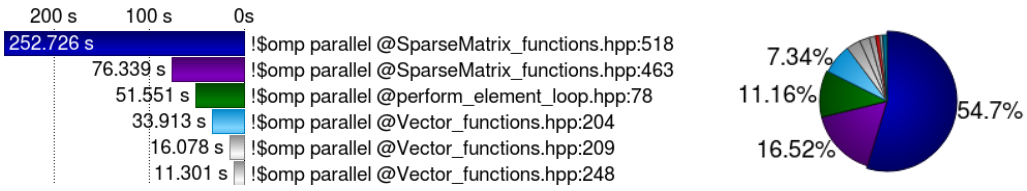
> *"[...] is a miniapp that mimics the finite element generation, assembly and solution for an unstructured grid problem. The physical domain is a 3D box with configurable dimensions and a structured discretization (which is treated as unstructured). The domain is decomposed using a recursive coordinate bisection (RCB) approach and the elements are simple hexahedra. The problem is linear and the resulting matrix is symmetric, so a standard conjugate gradient algorithm is used with a general sparse matrix data format and no preconditioning."*

MiniFE is part of the CORAL throughput benchmarks. It is written in C++ and makes use of templates to support different data types. I use the benchmark in the version 2.0-rc3, which is linked by the CORAL website. I compile the benchmark to utilize MPI and OpenMP for process and thread parallelization, respectively.
I visualize the execution of MiniFE on one compute node with two MPI ranks and twelve OpenMP threads per process in Figure 7.6. The timeline (Figure 7.6a) indicates that several phases are executed successively. Initially, the mesh is generated and filled, which is not parallel. Afterwards, the matrix structure is generated, which is thread sequential. In the first OpenMP parallel region, the finite element (FE) data is assembled (colored green). Then, the dirichlet boundary conditions are imposed (colored purple). Finally, the conjugate gradiants (CG) method is used to solve the given system. The functions with the highest runtime share in this phase are a matrix vector product (dark blue, defined in `SparseMatrix_functions.hpp` line 518) and the aggregation of two scaled vectors (BLAS function axpby, light blue, defined in `Vector_functions.hpp` line 169, includes parallel regions in lines 204 and 209). A closer look indicates a repetitive pattern for this phase (see Figure 7.6b). Figure 7.7 visualizes how runtime and power consumption of parallel regions that are executed in the matrix vector product and the vector aggregation functions relate to a changed configuration where the core frequency is reduced. Like in the previous section, I apply the least squares methods to calculate a fitting function for runtime and power, depending on the core frequency. Based on these, I calculate the function for energy consumption as a product of the former two.



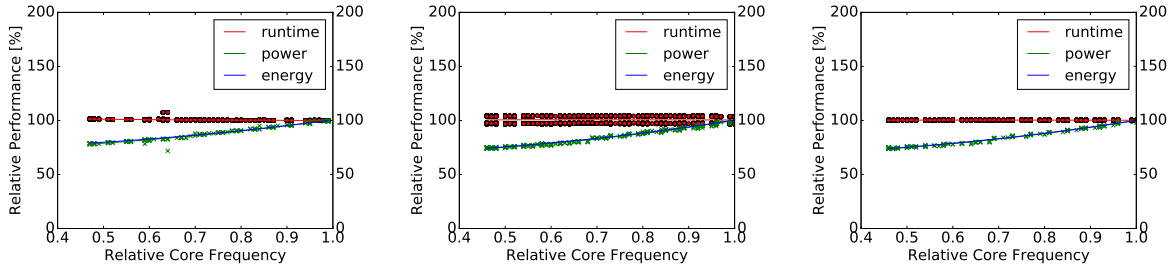
(a) Master Timeline: Two processes execute 12 OpenMP threads. The initialization phase is mostly thread serial. The later computation phase is thread parallel.

(b) Master Timeline: Zoom into conjugate-gradient solve phase.



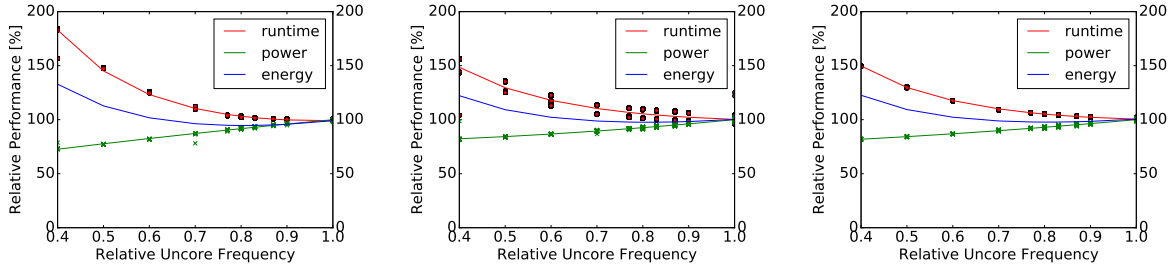
(c) Function Summary: Relative and absolute total runtime of instrumented functions

Figure 7.6: Vampir analysis of MiniFE benchmark. OpenMP parallel regions with a high share on the overall runtime are colored blue, purple, green and cyan. Other OpenMP regions are colored gray. MPI functions are marked red.



(a) SparseMatrix_functions.hpp:518 (b) Vector_functions.hpp:204 (c) Vector_functions.hpp:209

Figure 7.7: Scaling of performance, power, and energy with core frequency for selected parallel regions of MiniFE's CG phase. Points mark measured regions, lines show fitted functions.



(a) SparseMatrix_functions.hpp:518 (b) Vector_functions.hpp:204 (c) Vector_functions.hpp:209

Figure 7.8: Scaling of performance, power, and energy with uncore frequency for selected parallel regions of MiniFE's CG phase. Points mark measured regions, lines show fitted functions.

It is apparent that the runtime of none of the parallel regions that I analyze in this section scales with the core frequency. Instead, no performance loss can be observed when lowering the frequency. However, the power consumption is lowered significantly by up to 26 %. Therefore, a common core frequency of 1.2 GHz will be applied to the CG regions. Additional measurements have shown that the runtime of the initial regions increases with a lower core frequency. This is expected, since a significant time of the initialization is thread sequential and a single core is not able to utilize the main memory bandwidth to full capacity, as I have shown in Figure 4.22 on page 65. In such a scenario, the data paths within the processor become the bottleneck and any reduction of their performance lowers the overall throughput. Thus, the core frequency should be reduced at a late phase before the CG loop is started.

In a second step, I analyze how the uncore frequency affects runtime, power consumption and energy efficiency of the parallel regions. The appertaining plots are given in Figure 7.8. According to Figure 7.8a, the runtime of the matrix vector product increases by up to 85 % if the uncore frequency is reduced from 3 to 1.2 GHz. The runtime of the parallel regions that are part of the scaled vector aggregation is extended by 51 %. Apart from that, the power consumption savings are higher for the sparse matrix functions. Thus, a more suitable operation point in terms of ETS is 2.4 GHz.

In the third step of the analysis, I verify the assumed structure of the CG phase. The theory of a repetitive behavior that I mentioned earlier is proven by the event flow graph, which I show in Figure 7.9. Furthermore, according to this graph, the parallel region in `Vector_functions.hpp:204` is called in two different contexts within a single loop iteration of the CG phase. Therefore, there are two different runtime levels apparent in Figure 7.7b and Figure 7.8b. However, both have similar relative performance and power characteristics. Thus, a common configuration can be applied without capturing the context of the region, which would introduce another overhead source. Apart from that, there are four MPI synchronization points per loop iteration.

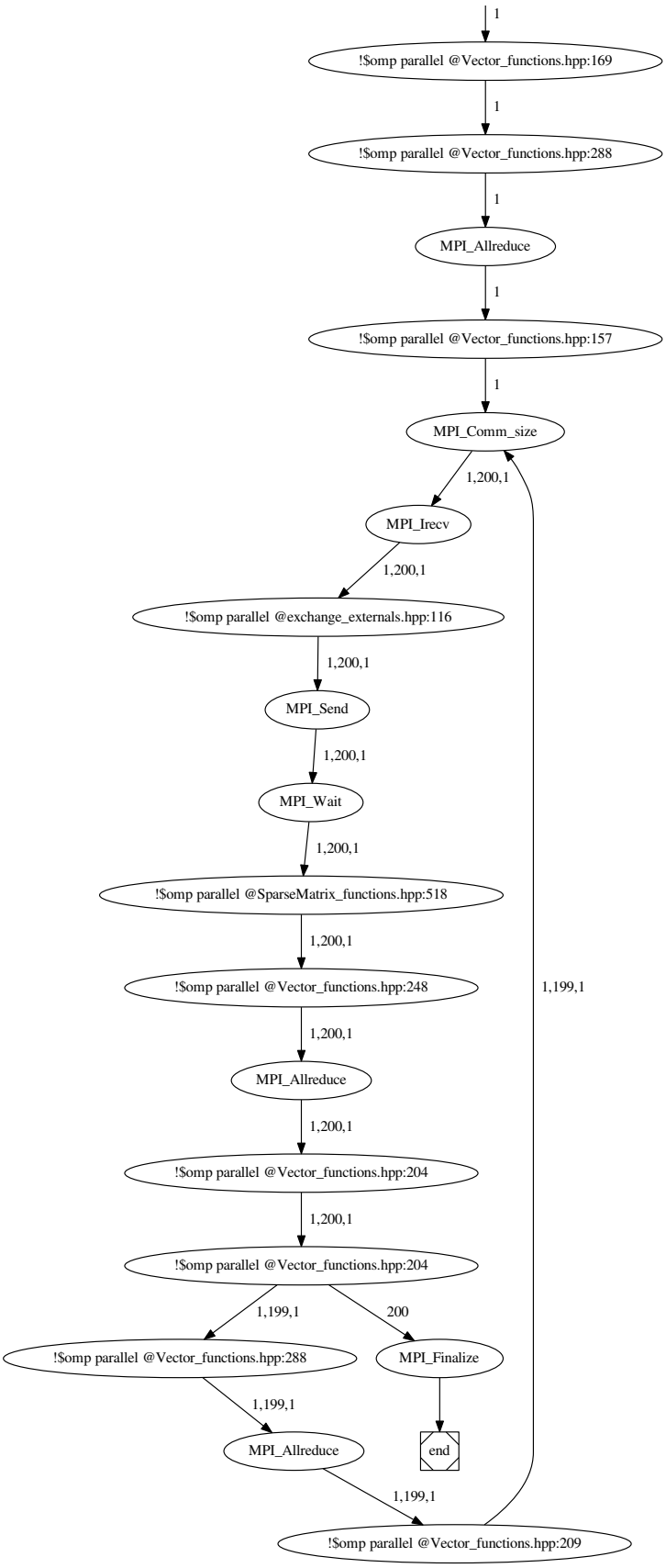


Figure 7.9: Leaf-node-EFG of the final phases of the MiniFE benchmark

```
1  name = ".*miniFE.x";
2  function_0:
3  {
4   name="!$omp parallel @Vector_functions.hpp:169";
5   dvfs_freq_after=1200000;
6   x86_adapt_Intel_UNCORE_MIN_RATIO_all_after=27;
7   x86_adapt_Intel_UNCORE_MAX_RATIO_all_after=27;
8  };
9  function_1:
10 {
11  name="!$omp parallel @SparseMatrix_functions.hpp:518";
12  x86_adapt_Intel_UNCORE_MIN_RATIO_all_before=24;
13  x86_adapt_Intel_UNCORE_MAX_RATIO_all_before=24;
14  x86_adapt_Intel_UNCORE_MIN_RATIO_all_after=27;
15  x86_adapt_Intel_UNCORE_MAX_RATIO_all_after=27;
16 };
```

Listing 7.2: Configuration file for optimizing MiniFE

Based on the analysis, I created the configuration file that is given in Listing 7.2. The configuration is changed at three different locations. Before the CG phase is entered, core and uncore frequency are reduced (lines 2–8). This is done after a specific parallel region that is not part of the loop is left[5]. Afterwards, the uncore frequency is reduced whenever the matrix vector product function is entered and reset afterwards (lines 9–16). This configuration file is used by the region-based tuning plugin, which lowers the respective frequencies and consequently the power consumption while running the CG phase. In Table 7.3, I depict the runtime as reported by the SPEC result file and energy efficiency results for different numbers of MPI ranks and OpenMP threads on 128 compute nodes of the taurus test system. One peculiarity is immediately apparent: the runtime of the benchmark increases with the proportion of thread parallelism. Even though the accompanying documentation states that the *"problem domain is initially decomposed using an RCB [recursive coordinate bisection] method which attempts to balance the subdomains"*, obviously the mechanism does not cover OpenMP parallelization. Therefore, the total runtime (and, consequently, the energy consumption) increases with a higher thread parallelism. However, two main objectives of the tuning are met. First, the runtime increases only marginally by less than 2 %. Second, the energy consumption is reduced. Depending on the segmentation between MPI ranks and OpenMP threads, energy consumption is reduced by 8.7-20.1 %.

Table 7.3: Runtime and energy consumption of medium sized MiniFE benchmark for default and tuned configuration

Number of MPI ranks	OpenMP threads per rank	runtime [s] uninstrumented min	**median**	max	tuned min	**median**	max	energy consumption [MJ] uninstrumented min	**median**	max	tuned min	**median**	max
3072	1	184	**189**	200	187	**189**	192	5.95	**6**	6.17	4.77	**4.79**	4.82
1536	2	185	**187**	188	188	**189**	190	5.81	**5.87**	5.91	4.73	**4.72**	4.73
768	4	220	**221**	221	222	**222**	223	6.46	**6.45**	6.46	5.33	**5.34**	5.32
512	6	247	**247**	248	250	**251**	253	6.87	**6.9**	6.93	5.8	**5.78**	5.84
384	8	273	**274**	280	275	**276**	277	7.35	**7.35**	7.47	6.19	**6.23**	6.24
256	12	343	**345**	346	346	**350**	374	8.51	**8.54**	8.54	7.44	**7.51**	7.93
128	24	707	**720**	722	710	**711**	740	14.41	**14.7**	14.72	13.42	**13.42**	13.9

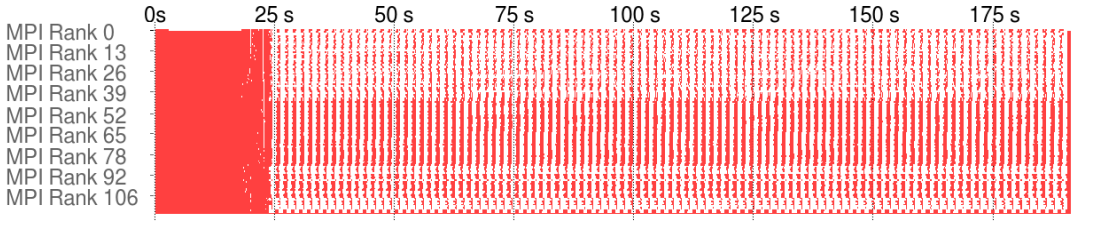
[5] I selected this function since it is the last parallel region prior to the CG loop that is not called at an earlier stage.

7.3 Balancing-based Tuning

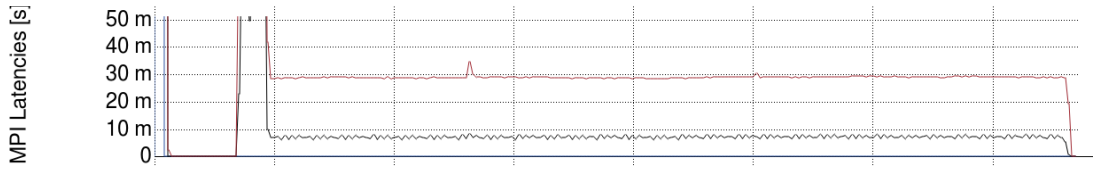
In this section, I demonstrate the effectiveness of the balancing-based online runtime tuning back end, which I described in Section 6.3.5.3, with two iterative solvers on the taurus HPC system. The solvers have an initialization phase, a processing phase, and a finalization phase. Both are parallel discrete-event simulationss (PDESs), which means that they model a system in parallel. The system has a specific state at each discrete time step. To translate the state of the system from one time step to another, each of the participating threads or processes computes the transitions for its share of the overall system. After the sub-states are calculated, they are synchronized so that in the next processing step the latest available data can be used. The computational demands for the single parts of the whole system can vary, e.g., if there is no cloud in one share of a modeled weather system but within another. In such a case some threads or processes arrive earlier in the synchronization then others and spend a significant amount of their time waiting for synchronization.

7.3.1 GemsFDTD

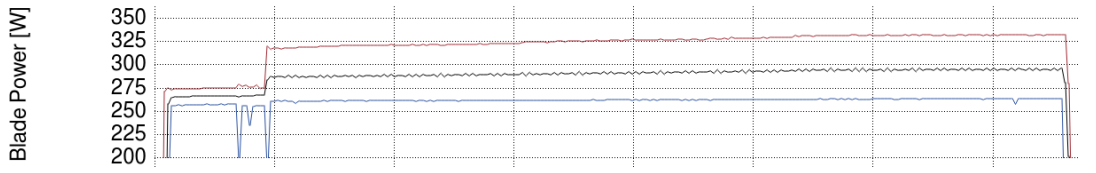
GemsFDTD stands for General ElectroMagnetic Solvers (GEMS) that are using finite-difference time-domain (FDTD) methods. The PDES computes a radar cross section of an object. It is written in Fortran90 and uses MPI as parallelization paradigm. Furthermore, GemsFDTD is part of the SPEC MPI2007 benchmark suite [MvWL+09, Section 3], which is used to evaluate the performance of parallel computing systems. SPEC MPI2007 provides standardized data sets that can be used depending on the size of the tested system. Figure 7.10 depicts the medium data set (mref) when the benchmark is executed on 120 MPI ranks. Due to restrictions of the benchmark, only 119 ranks are used, while one rank instantaneously finalizes calling `MPI_Finalize`. The remaining ranks execute an initialization phase until second 23. Afterwards, they process the single time steps (until second 190). Finally, they conclude the simulation, which takes about one second. The average node power consumption is approximately 270 Watt in the initialization and finalization phase and 290 W in the processing phase. Thus, in terms of time and energy, the processing phase should be target of the optimization. Additionally, the MPI Latencies that are calculated by Vampir as the average execution times of MPI calls



(a) Timeline

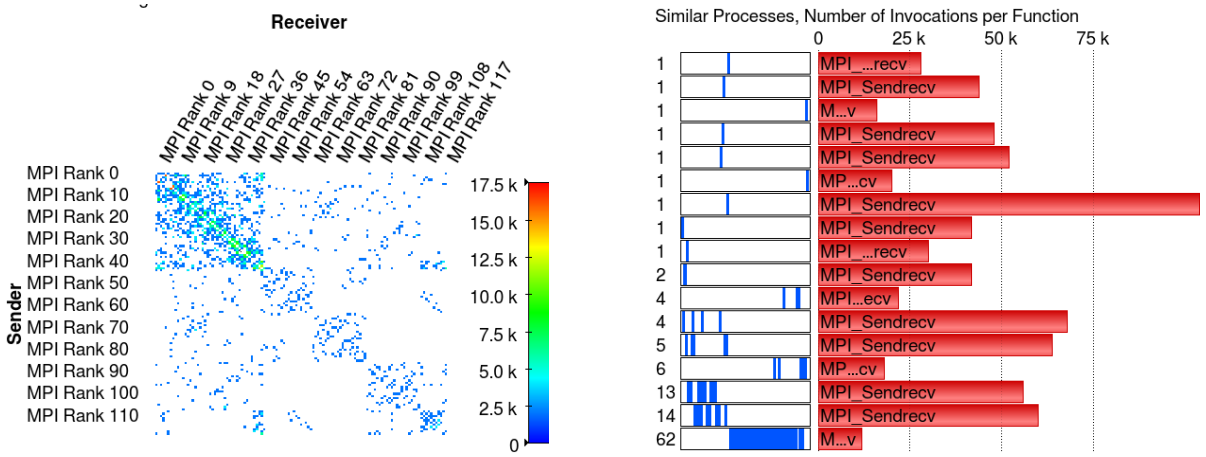


(b) Maximal (red), average (black), and minimal (blue) MPI latencies



(c) Maximal (red), average (black), and minimal (blue) blade power consumption

Figure 7.10: Vampir visualization of SPEC MPI benchmark 113.GemsFDTD (reference input set)



(a) Communication matrix of MPI ranks

(b) MPI ranks clustered by the number of MPI_Sendrecv calls

Figure 7.11: The number of calls to synchronization primitives differs between the single ranks. For example, 62 ranks call MPI_Sendrecv 11,964 times, one rank (43) calls it 103,688 times.

indicate that the respective region can be optimized. However, the load is not balanced between the single ranks as Figure 7.11 and Figure 7.12 show. Figure 7.11 describes the communication patterns of the single ranks within the processing phase (excluding the first and last time step, which call an additional `MPI_Barrier`). According to Figure 7.11a, the number of synchronizations in the first 48 ranks is significantly higher than for the other ranks. A more detailed clustering by Vampir (depicted in Figure 7.11b) shows that while one rank calls the used MPI synchronization function `MPI_Sendrecv` more than 103,688 times, most ranks call it only 11,964 times. Thus, the ranks do not execute a similar workload but have a different communication pattern. Additionally, the load imbalance leads to different execution times for the single synchronization calls. This is depicted in Figure 7.12, which visualizes a one second window of MPI communication. In this second, the average execution time for the synchronization region `MPI_Sendrecv` varies significantly between the ranks. While the total average of all processes is 2.254 ms, rank 48 waits 7.672 ms, and rank 108 waits only 1.251 ms. Thus, the program is certainly eligible to be optimized with the balancing plugin that is described in Section 6.3.5.3.

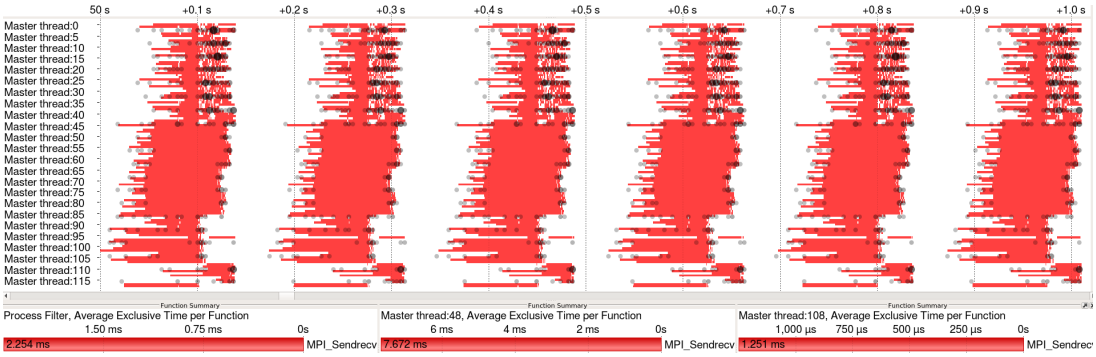
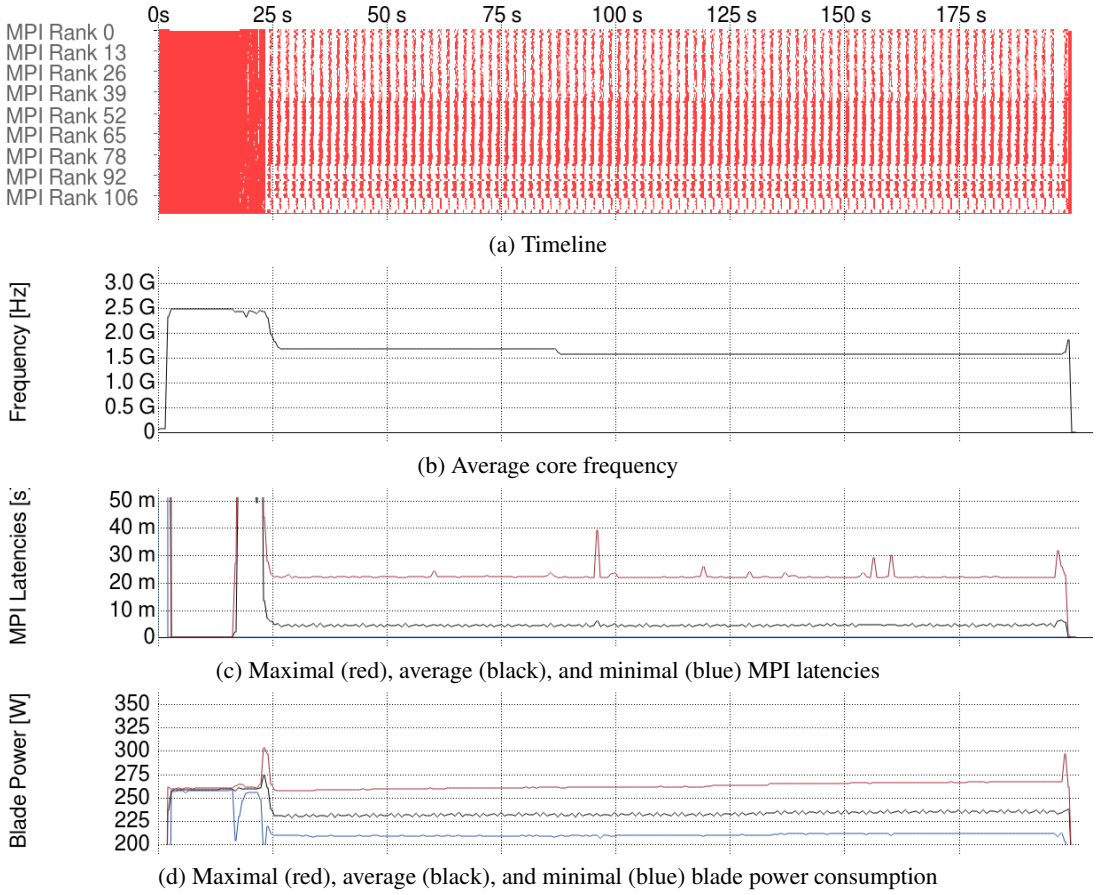


Figure 7.12: Vampir visualization of a 1 s segment of the processing phase of GemsFDTD. In the upper panel, the communication pattern of 119 of a total of 120 ranks is shown. Below, the average time of the single MPI calls is depicted, which is 2.254 ms over all processes and 7.672 ms and 1.251 ms for ranks 48 and 108, respectively. Both show that there is a significant imbalance between the individual MPI ranks.



(a) Timeline

(b) Average core frequency

(c) Maximal (red), average (black), and minimal (blue) MPI latencies

(d) Maximal (red), average (black), and minimal (blue) blade power consumption

Figure 7.13: Vampir visualization of an optimized run of the SPEC MPI2007 benchmark 113.GemsFDTD (medium reference input set). MPI latencies, some processor frequencies and the average power consumption decrease,while the runtime is extended.

A resulting trace of an optimized run is depicted in Figure 7.13. While the processor frequencies are reduced on ranks that are not on the critical path, the power consumption of the single nodes and the MPI Latencies are also lowered. However, the runtime is extended. To compare the effectiveness between the single variants, I run the benchmark without and with MPI instrumentation. The instrumented version calls either tracing, balancing, or both. Each of these variants is executed three times by providing the iterations argument to the `runspec` script used by the SPEC suites. I report runtimes and energy consumptions as reported by `sacct` in Table 7.4. According to the measurements, the runtime influence of instrumentation and tracing is negligible. The balancing adds a 4.1 % runtime overhead. However, the energy consumption is reduced by 14.3 %. The initial untuned results match the expected performance that is established by published results of comparable systems[6].

Table 7.4: Runtime and energy costs for Score-P runs of SPEC MPI2007 benchmark 113.GemsFDTD

Problem Size	Instrumentation and Substrates	runtime [s]			energy consumption [kJ]		
		min	**median**	max	min	**median**	max
medium	no instrumentation	192	**193**	202	268.6	**271.7**	281.0
	Score-P, Tracing	190	**191**	192	266.8	**270.7**	272.0
	Score-P, Balancing	201	**201**	203	232.5	**232.8**	233.7
	Score-P, Tracing + Balancing	200	**201**	204	233.0	**233.0**	235.6
large	Score-P, Tracing	319	**320**	322	941.0	**945.6**	940.5
	Score-P, Tracing + Balancing	319	**320**	320	801.1	**798.3**	800.0

[6] `http://spec.org/mpi2007/results/res2014q3/mpi2007-20140819-00465.txt`

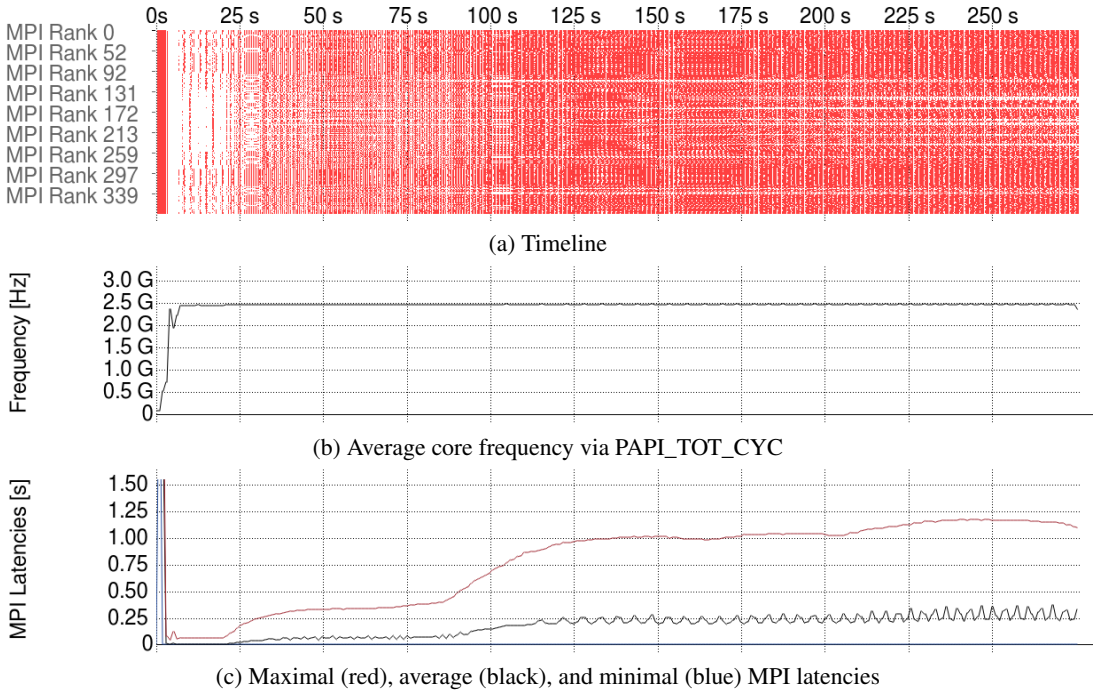
I verified the effectiveness by repeating the measurement with the large reference dataset and 240 MPI ranks. I skipped the executions that do not write a trace since the tracing overhead has been negligible before. Regardless of an activated balancing, the runtime is 320 seconds. However, energy consumption is reduced by 15.4 % from 945.6 kJ to 798.3 kJ and the average MPI Latencies in the processing phase are reduced from 15.3 to 6.2 ms.

7.3.2 COSMO SPECS+FD4

COSMO SPECS is a weather simulation. which consists of two parts: the Consortium for Small-scale Modeling (COSMO) and the SPECtral bin cloud microphysicS (SPECS) [GKS08]. Each of these parts processes different aspects of an atmospheric model. Thus, for each time step, COSMO and SPECS is executed. Between these sub-steps, the used data is translated to meet the requirements for the following sub-step. Lieber et al. describe that load imbalance accounts for more than 40 % of the overall runtime [LGW+12, Fig 4b]. They explain this with

> *“ Cloudy areas of the model domain [that] generate a substantially higher workload than cloudless areas. Such irregular workload variations require dynamic load balancing techniques . . . " ”*

Like the computational electromagnetics simulation GemsFDTD, COSMO SPECS uses Fortran90 as programming language and MPI as parallelization paradigm. Figure 7.14 shows a trace of COSMO SPECS on 384 MPI ranks. While the initialization time lasts about 3 seconds and the finalization approximately 1 second, the most significant share of the runtime (> 270 seconds) is spent in the processing phase. Within the processing phase, the time that is spent for MPI synchronizations increases with each time step. For example, the time spent within `MPI_Waitall` differs between 17 seconds on one rank and 185 seconds on another even though the number of calls are the same among all ranks. This is depicted in Figure 7.15. While the number of calls to blocking MPI functions provide a regular pattern (`MPI_Wait`, `MPI_Recv`) or are even called to an equal extend among all ranks (`MPI_Allreduce`,



(a) Timeline

(b) Average core frequency via PAPI_TOT_CYC

(c) Maximal (red), average (black), and minimal (blue) MPI latencies

Figure 7.14: Vampir visualization of COSMO SPECS

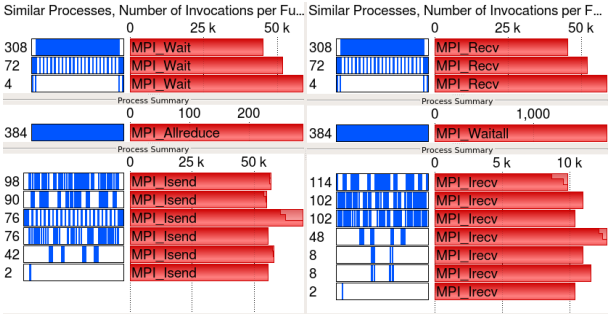
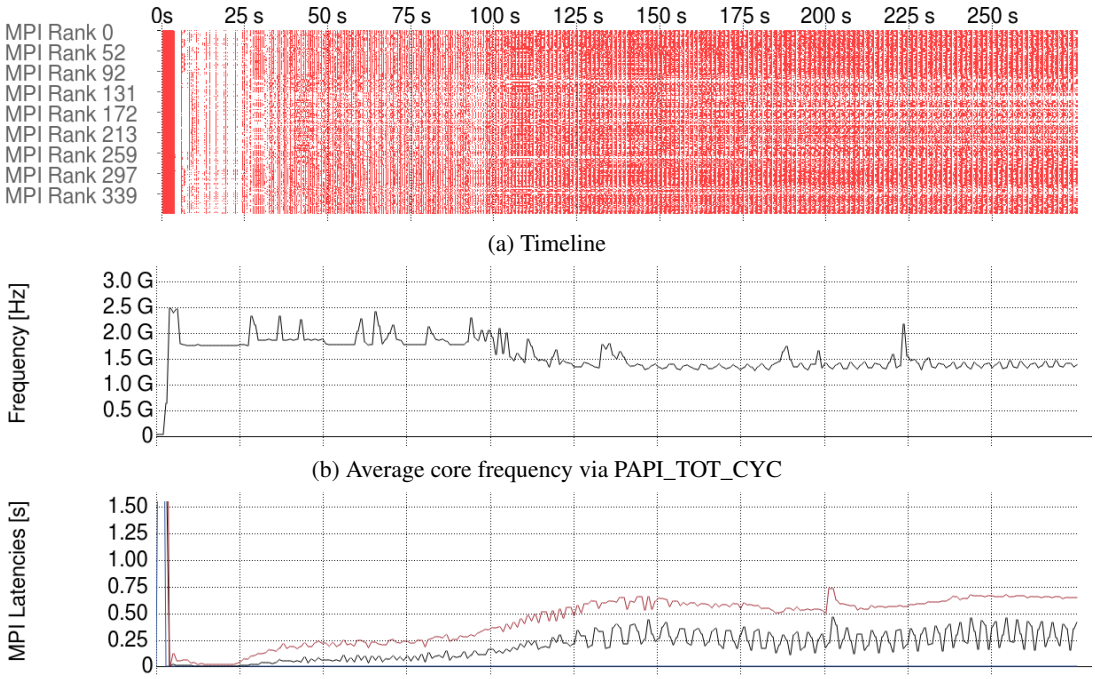


Figure 7.15: The number of calls to synchronization primitives differs between the single ranks. There are three different groups of MPI ranks for `MPI_Wait` and `MPI_Recv`. `MPI_Allreduce` and `MPI_Waitall` are called equally on all ranks.



(a) Timeline

(b) Average core frequency via PAPI_TOT_CYC

(c) Maximal (red), average (black), and minimal (blue) MPI latencies

Figure 7.16: Visualization of an optimized run of COSMO SPECS. In comparison to the initial situation (depicted in Figure 7.14), the frequencies off the critical path are reduced. This lowers the power consumption and the MPI latencies.

`MPI_Waitall`), the number of calls to non-blocking routines differs between the ranks. Thus, the communication pattern is also irregular, which is no problem for the balancing algorithm, though.
To verify the efficiency of the applied optimization, I run the simulation on 384 MPI ranks with and without balancing. The tracing is activated since its influence on the runtime is negligible. Again, each of these configurations is executed three times. The resulting runtimes and energy consumptions as reported by `sacct` are depicted in Table 7.5. While the runtime is increased by 1.4 %, the energy consumption is lowered by 19.3 %. Figure 7.16 shows a trace of the applied tuning where a reduction in frequency lowers the average MPI latencies.

Table 7.5: Runtime and energy costs for Score-P runs of COSMO SPECS

Instrumentation and Substrates	runtime [s]			energy consumption [kJ]		
	min	**median**	max	min	**median**	max
Score-P Tracing	276	**279**	283	1.17	**1.17**	1.2
Score-P Tracing + Balancing	282	**283**	287	938.9	**944.3**	953.7

7.4 Conclusion

In this chapter, I demonstrated that the infrastructure, which I introduced in Chapter 5 and whose implementation is described in Chapter 6, can be used to analyze the energy efficiency of HPC applications. Furthermore, I have shown that the created software can be used with the concepts given in Chapter 3 to increase the energy efficiency of parallel applications used in HPC by using power-saving mechanisms of contemporary processors. This is done in Section 7.2, where I demonstrated a region-based offline tuning back end. Based on a hardware and software analysis, specific regions are tuned for a low energy consumption, which improved the efficiency of selected workloads between 5.7 and 20.1 %. However, the proposed infrastructure also enables other energy efficiency tuning methods. In Section 7.3, I demonstrated the effectiveness of an online balancing-based energy efficiency tuning. This back end can be applied even if the single threads and processes execute different workloads, e.g., if there is an imbalance in the number of calls to synchronization libraries. While this tuning approach increased the runtime of the investigated workloads by 4.1 and 1.4 %, respectively, it also reduced the energy consumption by 14.3 and 19.3 %. By applying the two different energy efficiency tuning back end, I demonstrated that the infrastructure supports the most common parallelization paradigms (OpenMP and MPI) and programming languages (C++ and Fortran) used in HPC.

8 Summary, Outlook, and Future Work

"Don't adventures ever have an end? I suppose not. Someone else always has to carry on the story.", Bilbo Baggins
The Fellowship of the Ring by John Ronald Reuel Tolkien

8.1 Summary

In Chapter 3, I described a model that can be used to apply power-based energy efficiency tuning on HPC applications. This model includes hardware parameters that describe how fast specific power-saving methods can be applied and software parameters that define how a software is able to cope with a changed hardware configuration. I further described how event flow graphs and profiles can be used alternatively to timelines to gather the information necessary for an optimization, and I listed challenges and limitations for tuning.
In Chapter 4, I described the hardware parameters for common power-saving states and show how they can be measured. This includes a detailed analysis of the scope of ACPI states in Intel and AMD processors, the overhead to initiate a new state and the delay until the new state is applied. Furthermore, I described how the performance of memory bound workloads is characterized for low-power states. This information can be used to apply various energy efficiency optimization models and to fine-tune strategies that target an effective usage of power-saving mechanisms.
To capture software parameters and to apply tuning strategies, I decided to re-use existing performance-measurement infrastructures. This enabled me to intercept common programming languages and parallelization paradigms. I laid out a concept for a unified infrastructure in Chapter 5. First, I introduced an overview that distinguishes monitorable hardware and software items, their properties, and described their interaction. In a next step, I listed common components for a more general infrastructure and defined their behavior. Here, the software and hardware items are captured and used either for recording or tuning. I showed that this concept is applicable to various measurement tools, infrastructures, and tuning tools.
Based on this concept, I implemented an interface that can be used to augment traditional performance information with power and energy-related data. This interface, which I described in Section 6.2, enables users to collect software parameters for the proposed energy efficiency tuning model but also for other purposes that do not relate to energy efficiency. Researchers from various institutes have used this interface to analyze the efficiency of their workloads, as I described in Section 6.2.5. A second interface, which I described in Section 6.3, enables users to tune the energy efficiency of programs, alternatively to just measuring it. In the same section, I introduced three different plugins that can be used to analyze and tune the energy efficiency of HPC applications.
In the final chapter, I evaluated two tuning approaches by applying them to HPC benchmarks and applications. To do so, I used the instrumentation possibilities of Score-P and the different interfaces and plugins that I described earlier to re-act on specific software events and apply power-saving mechanisms. The first approach implements a region-based offline tuning that uses the model from Chapter 3 to determine energy-efficient configurations for program regions. Due to the selected evaluation function, the plugin prolonged the runtime by up to 4.6 %. However, the energy efficiency increased significantly by up to 20.1 %. The second plugin implements a balancing-based online tuning. It uses local information to evaluate the imbalance of parallel programs and applies more efficient configurations to lower the time that is spent in synchronization primitives. This lowered the energy consumption of the investigated workloads by up to 19.3 %, even though only local information is used.

8.2 Outlook

At processor level, multiple future trends can be seen, which already have consequences on contemporary processors. In [EBSA+11], Esmaeilzadeh et al. describe *dark silicon* as *"transistor under-utilization"*, which has two primary sources: parallelism and power. Based on modeled performances of the PARSEC benchmarks, they conclude that *"parallelism is the primary contributor to dark silicon"* for most of the benchmarks. However, if *"benchmarks have sufficient parallelism to even hypothetically sustain Moore's Law level speedup, [...] dark silicon due to power limitations constrains what can be realized"*, according to Esmaeilzadeh et al. This includes scaling applications that are used in HPC and the parallel execution of single threaded applications. Taylor [Tay12] lists three different alternatives that can take advantage of the higher performance and power dissipation density: shrinking processors, which for economical reasons he finds *"likely to happen only if we can find no practical use for dark silicon"*, *dim silicon*, and specialized co-processors. According to Taylor, dim silicon refers to logic *"that tr[ies] to retain general applicability across many applications"*. This is already incorporated and includes larger caches and turbo mechanisms that use the available power budget or even exceed it for a while.
The availability of new functional units and dim silicon spans a complex decision tree for processor architects (which new functions should be incorporated) as well as software architects (how can the available hardware mechanisms be exploited to run an application efficiently). Typically, processors are designed to run a variety of tasks efficiently to cover different fields of applications. Administrators and software developers can use hardware interfaces (BIOS and runtime tuning, see Section 2.5.3) to further configure the available mechanisms for the executed software. In lieu of the dark-silicon era, where it is to be expected that hardware becomes more configurable, the framework that I proposed supports software developers and performance analysts in multiple ways. First of all, the interface that is described in Section 6.2 can be used to include most different performance metrics into performance analysis tools. These can support the evaluation and pinpoint bottlenecks in the execution. Furthermore, they can be used to point out which components are used efficiently and which not. Based on a previous analysis, the interface discussed in Section 6.3 enables software engineers to configure the hardware and software environment dynamically, based on events within the executed software.
Not only on a processor but also on a data center scale, power budgets and power walls pose a problem. To tackle such limitations, Patki et al. suggest that one *"should also consider overprovisioning for high-performance computing (HPC)"* [PLR+13]. In such a scenario, *"full power [is guaranteed] to a restricted number of nodes (worst case provisioning)"* or *"power [is limited] to more nodes (overprovisioning)"*. However, such an approach needs a global infrastructure that constantly monitors the power consumption of components and shifts power budgets, as, for example, given in [PLR+16]. The interfaces that I designed and implemented can be used to support such an infrastructure in its decisions. Based on an offline analysis, the computing intensity and the consequential power consumption for single program regions could be provided as an input before the region is executed.

8.3 Future Work

Apart from energy efficiency, the described infrastructure also enables new options for performance analysis and debugging. One example has been the provision of event flow graphs (EFGs), which have already been used in this document to analyze the internal structure of parallel programs. Other back ends could visualize metrics and software event data live, write checkpoints at selected software events, or automatically change the concurrency in thread parallel programs using an online analysis.
A future development of the metric interface could enable more fine-grained hardware scopes in addition to "per-host" and "global". Such scopes depend on an extension of the Score-P *system tree* that describes the used hardware topology. Furthermore, additional data types could be included in addition to numeric ones. Based on requests by users, the substrate interface can be extended to provide new callbacks or new software events to possible plugins. The structure of the interfaces makes it easy to implement such extensions and enable new research in the areas of performance and energy efficiency evaluation and tuning.

Bibliography

[AAFP12] Victor Avelar, Dan Azevedo, Alan French, and Emerson Network Power. PUE: a comprehensive examination of the metric. Technical report, The Green Grid, 2012.

[AB09] Mikhail J. Atallah and Marina Blanton, editors. *Algorithms and Theory of Computation Handbook (Second Edition)*. Chapman and Hall/CRC, 2009. ISBN: 1584888202.

[ABF+10] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R. Tallent. HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6), 2010. DOI: 10.1002/cpe.1553.

[acp16] Advanced configuration and power interface (acpi) specification, revision 6.1, January 2016. online at uefi.org (accessed 2017-01-30).

[Adv13] Advanced Micro Devices. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, Rev 3.14*, Jan 2013. online at support.amd.com (accessed 2016-08-12).

[Adv15] Advanced Micro Devices. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 10h-1Fh Processors, Rev 3.12*, July 2015. online at support.amd.com (accessed 2016-08-12).

[Adv16] Advanced Micro Devices. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 60h-6Fh Processors, Rev 3.05*, May 2016. online at support.amd.com (accessed 2016-08-12).

[AFL14] Xavier Aguilar, Karl Fürlinger, and Erwin Laure. MPI trace compression using event flow graphs. *Euro-Par 2014 Parallel Processing*, 2014. DOI: 10.1007/978-3-319-09873-9_1.

[AKV+14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2014. DOI: 10.1145/2628071.2628092.

[AKVM07] Muhammad Ashraful Alam, Haldun Kufluoglu, Dhanoop Varghese, and Souvik Mahapatra. A comprehensive model for PMOS NBTI degradation: Recent progress. *Microelectronics Reliability*, 47(6), 2007. DOI: 10.1016/j.microrel.2006.10.012.

[AM05] Muhammad Ashrafu Alam and Souvik Mahapatra. A comprehensive model of PMOS NBTI degradation. *Microelectronics Reliability*, 45(1), Jan 2005. DOI: 10.1016/j.microrel.2004.03.019.

[AMD12] AMD. *AMD Opteron TM 6200 Series Processors Linux Tuning Guide*, apr 2012. Accessed online 2016-10-22.

[Ant14] Mikhail Yu. Antonov. *Computational Technologies: Advanced Topics*, chapter Architecture of parallel computing systems. De Gruyter, 2014. DOI: 10.1515/9783110359961.1.

[apm96] Intel Corporation, Microsoft Corporation. *Advanced Power Management (APM) BIOS Interface Specification, revision 1.2*, Feb 1996. online at microsoft.com (accessed 2017-01-30).

[Ass09] JEDEC Solid State Technology Association. JEDEC Standard – DDR2 SDRAM Specification, Nov 2009. JEDEC Document# JESD79-2F, online at jedec.org.

[Bar15] Barcelona Supercomputing Center (BSC). *Extrae tool user's Guide*, 3.2.1 edition, Nov 2015. online at tools.bsc.es (accessed 2017-02-12).

[BBB+94] David H. Bailey, Eric Barszcz, John Barton, David Browning, Russell Carter, Leo Dagum, Rod Fatoohi, Samuel Fineberg, Paul Frederickson, Thomas Lasinski, Rob Schreiber, Horst Simon, Venkat Venkatakrishnan, and Sisira Weeratunga. The NAS Parallel Benchmarks. Technical Report RNR-94-007, RNR, 1994. online at nas.nasa.gov (accessed 2017-01-05).

[BBS+00] David M. Brooks, Pradip Bose, Stanley E. Schuster, Hans Jacobson, Prabhakar N. Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish Gupta, and Peter W. Cook. Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors. *IEEE International Symposium on Microarchitecture (MICRO)*, 20(6), 2000. DOI: 10.1109/40.888701.

[BC10] Costas Bekas and Alessandro Curioni. A new energy aware performance metric. *Computer Science - Research and Development*, 25(3-4), Jul 2010. DOI: 10.1007/s00450-010-0119-z.

[BCOM+10] Franois Broquedis, Jerome Clet-Ortega, Stephanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. *18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2010. DOI: 10.1109/PDP.2010.67.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5), May 2000. DOI: 10.1145/358438.349303.

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *International Journal of High Performance Computing Applications*, 14(4), Nov 2000. DOI: 10.1177/109434200001400404.

[BKN+03] Shekhar Borkar, Tanay Karnik, Siva Narendra, Jim Tschanz, Ali Keshavarzi, and Vivek De. Parameter Variations and Impact on Circuits and Microarchitecture. In *Proceedings of the 40th annual Design Automation Conference*. ACM, 2003. DOI: 10.1145/775832.775920.

[BLFP10] Daniel Bedard, Min Yeol Lim, Robert Fowler, and Allan Porterfield. PowerMon: Fine-Grained and Integrated Power Monitoring for Commodity Computer Systems. In *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*, 2010. DOI: 10.1109/SECON.2010.5453824.

[BNRS13] Aurelien Bourdon, Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. PowerAPI: A Software Library to Monitor the Energy Consumed at the Processlevel. *ERCIM News*, 2013(92), 2013. online at https://hal.inria.fr (accessed 2017-03-01).

[BP06] Steven F. Barrett and Daniel J. Pack. *Microcontrollers Fundamentals for Engineers and Scientists*, volume 1. Morgan & Claypool Publishers, 2006. ISBN: 1598290584.

[BPG10] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. PERISCOPE: An Online-Based Distributed Performance Analysis Tool. *Tools for High Performance Computing 2009*, 2010. DOI: 10.1007/978-3-642-11261-4_1.

[BPP15] Sridutt Bhalachandra, Allan Porterfield, and Jan F. Prins. Using Dynamic Duty Cycle Modulation to Improve Energy Efficiency in High Performance Computing. *IEEE International Parallel and Distributed Processing Symposium (IPDPS) Workshops*, 2015. DOI: 10.1109/IPDPSW.2015.144.

[Bra07] Rodrigo Rubira Branco. Ltrace Internals. In *Proceedings of the Ottawa Linux Symposium (OLS)*, 2007. online at kernel.org (accessed 2017-01-10).

[BRPC07] Christian Belady, Andy Rawson, John Pfleuger, and Tahir Cader. Green grid data center power efficiency metrics: Pue and dcie (revision 2007-0). Technical report, The Green Grid, 2007.

[Bui06] Sang T Bui. Method and apparatus for performing wake on LAN power management, May 16 2006. US Patent 7,047,428.

[BWNH01] Holger Brunst, Manuela Winkler, Wolfgang E. Nagel, and Hans-Christian Hoppe. Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach. *Lecture Notes in Computer Science (LNCS)*, 2001. DOI: 10.1007/3-540-45718-6_80.

[CDG+93] David E Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten Von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing (SC)*. IEEE, 1993. DOI: 10.1145/169627.169724.

[CDKL14] Konstantinos Chasapis, Manuel Dolz, Michael Kuhn, and Thomas Ludwig. Evaluating Power-Performace Benefits of Data Compression in HPC Storage Servers. In *4th International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies (ENERGY 2014)*. IARIA XPS Press, 2014.

[CM05] Gilberto Contreras and Margaret Martonosi. Power Prediction for Intel XScale Processors Using Performance Monitoring Unit Events. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2005. DOI: 10.1145/1077603.1077657.

[CMBAN08] Matthew Curtis-Maury, Filip Blagojevic, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Prediction-Based Power-Performance Adaptation of Multithreaded Scientific Codes. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 19(10), 2008. DOI: 10.1109/TPDS.2007.70804.

[CMSB+08] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Prediction Models for Multi-Dimensional Power-Performance Optimization on Many Cores. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT)*. ACM, 2008. DOI: 10.1145/1454115.1454151.

[CSB92] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low-power CMOS Digital Design. *IEEE Journal of Solid-State Circuits*, 27(4), 1992. DOI: 10.1109/4.126534.

[CTC14] Pietro Cicotti, Anish Tiwari, and Laura Carrington. Efficient speed (ES): Adaptive DVFS and Clock Modulation for Energy Efficiency. In *IEEE International Conference on Cluster Computing (CLUSTER),*. IEEE, 2014. DOI: 10.1109/CLUSTER.2014.6968750.

[Dar01] Frederica Darema. The SPMD Model: Past, Present and Future. *Lecture Notes in Computer Science (LNCS)*, 2001. DOI: 10.1007/3-540-45417-9_1.

[DE98] Leonardo Dagum and Rameshm Enon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1), 1998. DOI: 10.1109/99.660313.

[DGH+10] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanaa, and Christian Le. RAPL: Memory Power Estimation and Capping. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 189–194. ACM, 2010. DOI: 10.1145/1840845.1840883.

[DHK+15] Manuel F. Dolz, Mohammad Reza Heidari, Michael Kuhn, Thomas Ludwig, and German Fabregat. ARDUPOWER: A low-cost wattmeter to improve energy efficiency of HPC applications. *Sixth International Green and Sustainable Computing Conference (IGSC)*, Dec 2015. DOI: 10.1109/IGCC.2015.7393692.

[DM02] Patrick G. Drennan and Colin C. McAndrew. Understanding MOSFET Mismatch for Analog Design. In *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC)*, 2002. DOI: 10.1109/CICC.2002.1012872.

[DvdS12] Jack J. Dongarra and Aad J. van der Steen. High-performance computing systems: Status and outlook. *Acta Numerica*, 21, 5 2012. DOI: 10.1017/S0962492912000050.

[dWJ03] Rob F. Van der Wijngaart and Haoqiang Jin. NAS Parallel Benchmarks, Multi-Zone Versions. Technical Report NAS-03-010, NASA Advanced Supercomputing Division (NAS), 2003. online at nas.nasa.gov (accessed 2017-01-05).

[EBSA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011. 10.1145/2000064.2000108.

[FAWR+11] Michael Floyd, Malcolm Allen-Ware, Karthick Rajamani, Bishop Brock, Charles Lefurgy, Alan J. Drake, Lorena Pesantez, Tilman Gloekler, Jose A. Tierno, Pradip Bose, and et al. Introducing the Adaptive Energy Management Features of the Power7 Chip. *Proceedings of the IEEE International Symposium on Microarchitecture (MICRO)*, 31(2), Mar 2011. DOI: 10.1109/MM.2011.29.

[FCGS14] Gilles Fourestey, Ben Cumming, Ladina Gilly, and Thomas C. Schulthess. First Experiences with Validating and Using the Cray Power Management Database Tool. *Cray User Group*, 2014. online at cug.org (accessed 2017-01-30).

[FJ05] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 2005. DOI: 10.1109/JPROC.2004.840301.

[Fly72] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers (TC)*, C-21(9), Sep 1972. DOI: 10.1109/TC.1972.5009071.

[FP02] Joel H. Ferziger and Milovan Peric. *Computational Methods for Fluid Dynamics*. Springer-Verlag Berlin Heidelberg, 3 edition, 2002. DOI: 10.1007/978-3-642-56026-2.

[Fre16] Free Software Foundation, 51 Franklin Street, Fifth Floor Boston, MA 02110-1301, USA. *GNU Offloading and Multi Processing Runtime Library - GCC*, 6.1 edition, 2016. online at gcc.gnu.org (accessed 2016-06-30).

[FS09] Wu-Chun Feng and Thomas Scogland. The Green500 list: Year One. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2009. DOI: 10.1109/IPDPS.2009.5160978.

[GBCH01] Stephen Gunther, Frank Binns, Douglas M. Carmean, and Jonathan C. Hall. Managing the Impact of Increasing Microprocessor Power Consumption. *Intel Technology Journal*, 5(1), 2001.

[GCB15] Michael Gerndt, Eduardo César, and Siegfried Benkner. *Automatic Tuning of HPC Applications: The Periscope Tuning Framework*. Shaker, 1 edition, 2015. ISBN: 3844035176.

[GFB+04] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodal. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2004. DOI: 10.1007/978-3-540-30218-6_19.

[GFS+10] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W. Cameron. PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 21(5):658–671, 2010. DOI: 10.1109/TPDS.2009.76.

[GHBDL09] Anshul Gandhi, Mor Harchol-Balter, Rajarshi Das, and Charles Lefurgy. Optimal Power Allocation in Server Farms. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 157–168. ACM, 2009. DOI: 10.1145/1555349.1555368.

[GJW11] Brian Guenter, Navendu Jain, and Charles Williams. Managing Cost, Performance, and Reliability Tradeoffs for Energy-Aware Server Provisioning. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM),*. IEEE, 2011. DOI: 10.1109/INFCOM.2011.5934917.

[GKL99] Christos John Georgiou, Edward Scott Kirkpatrick, and Thor Arne Larsen. Performance-temperature optimization by cooperatively varying the voltage and frequency of a circuit, August 17 1999. US Patent 5,940,785.

[GKS08] Verena Grützun, Oswald Knoth, and Martin Simmel. Simulation of the influence of aerosol particle characteristics on clouds and precipitation with LM-SPECS: Model description and first results. *Atmospheric Research*, 90(2-4):233–242, Nov 2008. DOI: 10.1016/j.atmosres.2008.03.002.

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel computing*, 22(6):789–828, 1996. DOI: 10.1016/0167-8191(96)00024-5.

[GMG+10] Bhavishya Goel, Sally A McKee, Roberto Gioiosa, Karan Sing, Major Bhadauria, and Marco Cesati. Portable, Scalable, per-Core Power Estimation for Intelligent Resource Management. In *International Green Computing Conference (IGCC)*, pages 135–146. IEEE, 2010. DOI: 10.1109/GREENCOMP.2010.5598313.

[GSS15] Corey Gough, Ian Steiner, and Winston A. Saunders. *Energy Efficient Servers: Blueprints for Data Center Optimization*. Apress, 2015. ISBN: 1430266376.

[GWW+10] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6), April 2010. DOI: 10.1002/cpe.1556.

[Hal06] Christopher Hallinan. *Embedded Linux Primer: A Practical Real-World Approach*. Prentice Hall, 2006. ISBN: 9780137017836.

[HBB+10] David L. Hill, Derek Bachand, Selim Bilgin, Robert Greiner, Per Hammarlund, Thomas Huff, Steve Kulick, and Robert Safranek. The Uncore: A Modular Approach to Feeding the High-performance Cores. *Intel Technology Journal*, 14(3), 2010. online at intel.com (accessed 2016-10-20).

[HDC+09] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. Improving Performance via Mini-applications. Sandia report, Sandia National Laboratories, 2009. online at mantevo.org (accessed 2017-01-05).

[HDVH12] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring Energy Consumption for Short Code Paths Using RAPL. *SIGMETRICS Performance Evaluation Review*, 40(3), Jan 2012. DOI: 10.1145/2425248.2425252.

[HF16] Johannes Hofmann and Dietmar Fey. An ECM-based Energy-efficiency Optimization Approach for Bandwidth-limited Streaming Kernels on Recent Intel Xeon Processors. In *Proceedings of the 4th International Workshop on Energy Efficient Supercomputing (E2SC)*. IEEE, 2016. DOI: 10.1109/E2SC.2016.16.

[HGS98] A. Ira Horden, Steven D. Gorman, and Lionel S. Smith. Method and apparatus providing multiple voltages and frequencies selectable based on real time criteria to control power consumption, Sep 1998. US Patent 5,812,860.

[HIG94] Mark Horowitz, Thomas Indermaur, and Ricardo Gonzalez. Low-power digital design. *IEEE Symposium on Low Power Electronics, Digest of Technical Papers*, 1994. DOI: 10.1109/LPE.1994.573184.

[HIS+13] Daniel Hackenberg, Thomas Ilsche, Robert Schöne, Daniel Molka, Maik Schmidt, and Wolfgang E. Nagel. Power Measurement Techniques on Standard Compute Nodes: A Quantitative Comparison. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013. DOI: 10.1109/ISPASS.2013.6557170.

[HIS+14] Daniel Hackenberg, Thomas Ilsche, Joseph Schuchart, Robert Schöne, Wolfgang E Nagel, Marc Simon, and Yiannis Georgiou. HDEEM: High Definition Energy Efficiency Monitoring. In *Energy Efficient Supercomputing Workshop (E2SC)*. IEEE, 2014. DOI: 10.1109/E2SC.2014.13.

[HM08] Serge E. Hallyn and Andrew G. Morgan. Linux capabilities: Making them work. In *Proceedings of the Ottawa Linux Symposium (OLS)*, volume 8, 2008. online at http://kernel.org (accessed 2017-01-12).

[HMdS+12] Tobias Hilbrich, Matthias S. Müller, Bronis R. de Supinski, Martin Schulz, and Wolfgang E. Nagel. GTI: A Generic Tools Infrastructure for Event-Based Tools in Parallel Systems. *IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, May 2012. DOI: 10.1109/IPDPS.2012.123.

[HOMS13] Daniel Hackenberg, Roland Oldenburg, Daniel Molka, and Robert Schöne. Introducing FIRESTARTER: A Processor Stress Test Utility. In *International Green Computing Conference (IGCC)*. IEEE, June 2013. DOI: 10.1109/IGCC.2013.6604507.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011. ISBN: 012383872X.

[HRD+14] Alistair Hart, Harvey Richardson, Jens Doleschal, Thomas Ilsche, Mario Bielert, and Matthew Kappel. User-level Power Monitoring and Application Performance on Cray XC30 Supercomputers. In *Proceedings of the Cray User Group*, 2014. online at cug.org (accessed 2017-01-30).

[HSI+15] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An Energy Efficiency Feature Survey of the Intel Haswell Processor. In *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*. IEEE, 2015. DOI: 10.1109/IPDPSW.2015.70.

[HSM+10] Daniel Hackenberg, Robert Schöne, Daniel Molka, Matthias S. Müller, and Andreas Knüpfer. Quantifying power consumption variations of HPC systems using SPEC MPI benchmarks. *Computer Science - Research and Development*, 25(3-4), Jul 2010. DOI: 10.1007/s00450-010-0118-0.

[IHG+15] Thomas Ilsche, Daniel Hackenberg, Stefan Graul, Robert Schöne, and Joseph Schuchart. Power Measurements for Compute Nodes: Improving Sampling Rates, Granularity and Accuracy. *Sixth International Green and Sustainable Computing Conference (IGCC)*, 2015. DOI: 10.1109/IGCC.2015.7393710.

[IM03] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: methodology and empirical data. *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003. DOI: 10.1109/MICRO.2003.1253186.

[Int09] Intel. *Intel Core i7-800 and i5-700 Desktop Processor Series and LGA1156 Socket Thermal/Mechanical Specifications and Design Guidelines*, 2009. online at Intel.com (accessed 2016-08-05).

[Int10] Intel Corporation. *Intel Xeon 5600 Series, Datasheet, Volume Two*, Mar 2010. online at intel.com (accessed 2016-08-12).

[Int11a] Intel Corporation. *2nd Generation Intel Core Processor Family Desktop and Intel Pentium Processor Family Desktop, and LGA1155 Socket Thermal Mechanical Specifications and Design Guidelines (TMSDG)*, May 2011. online at intel.com (accessed 2016-08-12).

[Int11b] Intel Corporation. *2nd Generation Intel Core Processor Family Desktop, Datasheet – Volume 2*, Sept 2011. online at intel.com (accessed 2016-08-12).

[Int11c] Intel Corporation. *Intel Core i5-600, i3-500 Desktop Processor Series, Intel Pentium Desktop Processor 6000 Series Datasheet – Volume 1*, Jan 2011. online at intel.com (accessed 2016-08-12).

[Int11d] Intel Corporation. *Intel Xeon 5600 Series, Datasheet, Volume One*, Jun 2011. online at intel.com (accessed 2016-08-12).

[Int12a] Intel. *Intel Xeon Processor E5-2600 Product Family Uncore Performance Monitoring Guide*, Mar 2012. Reference Number: 327043-001, online at intel.com (accessed 2017-02-02).

[Int12b] Intel Corporation. *Intel Xeon Processor E5-1600/E5-2600/E5-4600 Product Families, Datasheet – Volume One*, May 2012. online at intel.com (accessed 2016-08-12).

[Int12c] Intel Corporation. *Intel Xeon Processor E5-1600/E5-2600/E5-4600 Product Families, Datasheet – Volume Two*, May 2012. online at intel.com (accessed 2016-08-12).

[Int13a] Intel. *Desktop 3rd Generation Intel Core Processor Family, Desktop Intel Pentium Processor Family, Desktop Intel Celeron Processor Family, and LGA1155 Socket Thermal Mechanical Specifications and Design Guidelines (TMSDG)*, 2013. online at Intel.com (accessed 2016-08-05).

[Int13b] Intel Corporation. *2nd Generation Intel Core Processor Family Desktop, Datasheet – Volume 1*, Jun 2013. online at intel.com (accessed 2016-08-12).

[Int13c] Intel Corporation. *3rd Generation Intel CoreTM Processor Family Desktop and Intel Pentium Processor Family Desktop, and LGA1155 Socket Thermal Mechanical Specifications and Design Guidelines (TMSDG)*, Jan 2013. online at intel.com (accessed 2016-08-12).

[Int13d] Intel Corporation. *Desktop 3rd Generation Intel Core Processor Family, Desktop Intel Pentium Processor Family, and Desktop Intel Celeron Processor Family, Datasheet – Volume 1 of 2*, Nov 2013. online at intel.com (accessed 2016-08-12).

[Int13e] Intel Corporation. *Desktop 3rd Generation Intel Core Processor Family, Desktop Intel Pentium Processor Family, and Desktop Intel Celeron Processor Family, Datasheet – Volume 2 of 2*, Nov 2013. online at intel.com (accessed 2016-08-12).

[Int13f] Intel Corporation. *Desktop 4th Generation Intel Core TM Processor Family, Desktop Intel Pentium Processor Family, Desktop Intel Celeron Processor Family, and Intel Xeon Processor E3-1200 v3 Product Family*, Dec 2013. online at intel.com (accessed 2016-08-12).

[Int14a] Intel Corporation. *Desktop 4th Generation Intel Core Processor Family, Desktop Intel Pentium Processor Family, and Desktop Intel Celeron Processor Family, Datasheet – Volume 1 of 2*, Mar 2014. online at intel.com (accessed 2016-08-12).

[Int14b] Intel Corporation. *Desktop 4th Generation Intel Core Processor Family, Desktop Intel Pentium Processor Family, and Desktop Intel Celeron Processor Family, Datasheet – Volume 2 of 2*, Mar 2014. online at intel.com (accessed 2016-08-12).

[Int14c] Intel Corporation. *Intel Xeon Processor E5-1600/E5-2600/E5-4600 v2 Product Families, Datasheet – Volume One of Two*, Mar 2014. online at intel.com (accessed 2016-08-12).

[Int15a] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A, 3B, and 3C: System Programming Guide*, Dec 2015. Order Number: 325384-057US, online at intel.com (accessed 2017-02-12).

[Int15b] Intel. *Intel Xeon Processor E5-1600/2400/2600/4600 v3 Product Families Datasheet - Volume 1: Electrical Datasheet*, Jun 2015. online at intel.com (accessed 2016-08-12).

[Int15c] Intel. *Intel Xeon Processor E5-1600/2400/2600/4600 v3 Product Families Datasheet - Volume 2: Registers*, Jun 2015. online at intel.com (accessed 2016-08-12).

[Int15d] Intel. *Intel Xeon Processor E5 and E7 v3 Family Uncore Performance Monitoring Reference Manual*, Jun 2015. Reference Number: 331051-002, online at intel.com (accessed 2016-10-20).

[Int15e] Intel. *Intel(R) MPI Library Reference Manual for Linux* OS*, jul 2015. Document Number: 315399-011US, online at intel.com (accessed 2017-01-30).

[Int16a] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Jun 2016. Order Number: 248966-033, online at intel.com (accessed 2017-01-30).

[Int16b] Intel Corporation. *5th Generation Intel Core Processor Family, Intel Core M Processor Family, Mobile Intel Pentium Processor Family, and Mobile Intel Celeron Processor Family, Volume 1 of 2*, May 2016. online at intel.com (accessed 2017-01-12).

[Int16c] Intel Corporation. *6th Generation Intel Processor Datasheet for S-Platforms Datasheet, Volume 1 of 2*, May 2016. online at intel.com (accessed 2016-08-12).

[Int16d] Intel Corporation. *6th Generation Intel Processor Datasheet for S-Platforms Datasheet, Volume 2 of 2*, Feb 2016. online at intel.com (accessed 2016-08-12).

[Int16e] Intel Corporation. *Intel Math Kernel Library for Linux OS Developer Guide*, intel mkl 11.3 - linux os revision: 052 edition, Apr 2016. online at intel.com (accessed 2017-01-30).

[Int16f] Intel Corporation. *Intel MPI Library for Linux OS Developer Guide*, 2016. online at intel.com (accessed 2017-01-17).

[Int16g] Intel Corporation. *Intel OpenMP Runtime Library*, Aug 2016. online at openmprtl.org (accessed 2017-01-30).

[ISO14] C++ international standard. Standard N3337, ISO International Standard Organization, JTC1/SC22/WG21 - The C++ Standards Committee, 2014. draft online at open-std.org (accessed 2017-01-30).

[ISSH15] Thomas Ilsche, Joseph Schuchart, Robert Schöne, and Daniel Hackenberg. Combining Instrumentation and Sampling for Trace-based Application Performance Analysis. In *Tools for High Performance Computing 2014*, pages 123–136. Springer, 2015. DOI: 10.1007/978-3-319-16012-2_6.

[Jai91] Raj K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991. ISBN: 0471503363.

[JBC+15] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John Stratton, Alexey Titov, Ke Wang, Matthijs van Waveren, Brian Whitney, Sandra Wienke, Rengan Xu, and Kalyan Kumaran. SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance. *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, 2015. DOI: 10.1007/978-3-319-17248-4_3.

[JBOO15] Christopher January, Jonathan Byrd, Xavier Oró, and Mark O'Connor. Allinea MAP: Adding Energy and OpenMP Profiling Without Increasing Overhead. In *Tools for High Performance Computing 2014*. Springer, 2015. DOI: 10.1007/978-3-319-16012-2_2.

[JM01] Russ Joseph and Margaret Martonosi. Run-time Power Estimation in High Performance Microprocessors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design (ISPLED)*. ACM, 2001. DOI: 10.1145/383082.383119.

[JSK+10] Ravi Jotwani, Sriram Sundaram, Stephen Kosonocky, Alex Schaefer, Victor Andrade, Greg Constant, Amy Novak, and Samuel Naffziger. An x86-64 Core Implemented in 32nm SOI CMOS. In *IEEE International Solid-State Circuits Conference (ISSCC)*, 2010. DOI: 10.1109/ISSCC.2010.5434076.

[Juc12] Guido Juckeland. *Trace-based Performance Analysis for Hardware Accelerators*. PhD thesis, TU Dresden, 2012. online at qucosa.de (accessed 2017-01-30).

[KFH+14] Michael Knobloch, Maciej Foszczynski, Willi Homberg, Dirk Pleiter, and Hans Böttiger. Mapping fine-grained power measurements to HPC application runtime characteristics on IBM POWER7. *Computer Science - Research and Development*, 29, 2014. DOI: 10.1007/s00450-013-0245-5.

[KFMPN12] Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn, and Wolfgang E. Nagel. Scout: A Source-to-Source Transformator for SIMD-Optimizations. *Lecture Notes in Computer Science (LNCS)*, 2012. DOI: 10.1007/978-3-642-29740-3_17.

[KH03] Steve Kaufmann and Bill Homer. CrayPat-Cray X1 Performance Analysis Tool. *Proceedings of the Cray User Group*, 2003. online at cug.org (accessed 2017-03-07).

[KHN12] Michael Kluge, Daniel Hackenberg, and Wolfgang E. Nagel. Collecting Distributed Performance Data with Dataheap: Generating and Exploiting a Holistic System View. *Proceedings of the International Conference on Computational Science (ICCS)*, 2012. DOI: 10.1016/j.procs.2012.04.215.

[Kle05] Andi Kleen. A NUMA API for Linux. *Novel Inc.*, 2005. online at halobates.de (accessed 2017-01-30).

[KMPP07] Jim Keniston, Ananth Mavinakayanahalli, Prasanna Panchamukhi, and Vara Prasad. Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps. In *Proceedings of the Ottawa Linux Symposium (OLS)*, 2007. online at kernel.org (accessed 2017-01-12).

[KRaM+12] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, et al. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*. Springer, 2012. DOI: 10.1007/978-3-642-31476-6_7.

[Kri05] R Krishnakumar. Kernel Korner: Kprobes–A Kernel Debugger. *Linux Journal*, 2005(133), 2005.

[Lam13] Christoph Lameter. NUMA (Non-Uniform Memory Access): An Overview. *Queue*, 11(7), Jul 2013. DOI: 10.1145/2508834.2513149.

[Lan09] Klaus-Dieter Lange. Identifying Shades of Green: The SPECpower Benchmarks. *IEEE Computer*, 42(3), 2009. DOI: 10.1109/MC.2009.84.

[LC08] Ewing Lusk and Anthony Chan. Early Experiments with the OpenMP/MPI Hybrid Programming Model. *Lecture Notes in Computer Science (LNCS)*, 2008. DOI: 10.1007/978-3-540-79561-2_4.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2005. DOI: 10.1145/1065010.1065034.

[LDGB+15] Pablo Llopis, Manuel F. Dolz, Javier García-Blas, Florin Isaila, Jesús Carretero, Mohammad Reza Heidari, and Michael Kuhn. Analyzing Power Consumption of I/O Operations in HPC Applications. In *Proceedings of the Second International Workshop on Sustainable Ultrascale Computing Systems (NESUS)*, pages 107–116. Computer Architecture, Communications and Systems Group (ARCOS), 2015.

[LDS+93] Sandra Loosemore, Ulrich Drepper, Richard M Stallman, Andrew Oram, and Roland McGrath. *The GNU C Library Reference Manual*, 1993. online at gnu.org (accessed 2017-01-30).

[Lev03] John Levon. Oprofile internals. *OProfile online documentation*, 2003. online at sourceforge.net (accessed 2017-03-07).

[Lev04] John Levon. Oprofile manual. *OProfile online documentation*, 2004. online at sourceforge.net (accessed 2017-03-07).

[LGW+12] Matthias Lieber, Verena Grützun, Ralf Wolke, Matthias S. Müller, and Wolfgang E. Nagel. Highly Scalable Dynamic Load Balancing in the Atmospheric Modeling System COSMO-SPECS+FD4. *Lecture Notes in Computer Science (LNCS)*, 2012. DOI: 10.1007/978-3-642-28151-8_13.

[Lov13] Robert Love. *Linux system programming: talking directly to the kernel and C library*. O'Reilly Media, Inc., 2013. ISBN: 1449339530.

[LOW96] Thomas Ludwig, Michael Oberhuber, and Roland Wismüller. An Open Monitoring System for Parallel and Distributed Programs. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Parallel Processing: Second International Euro-Par Conference Proceedings, Volume I*. Springer Berlin Heidelberg, 1996. DOI: 10.1007/3-540-61626-8_9.

[LPD13] James H. Laros III, Phil Pokorny, and David Debonis. PowerInsight - A Commodity Power Measurement Capability. In *International Green Computing Conference (IGCC)*, 2013. DOI: 10.1109/IGCC.2013.6604485.

[Luc08] Dinh The Luc. Pareto optimality. *Springer Optimization and Its Applications*, 2008. DOI: 10.1007/978-0-387-77247-9_18.

[LYdS+13] Chunhua Liao, Yonghong Yan, Bronis R de Supinski, Daniel J Quinlan, and Barbara Chapman. Early Experiences With The OpenMP Accelerator Model. In *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 2013. DOI: 10.1007/978-3-642-40698-0_7.

[MBB+12] Matthias S. Müller, John Baron, William C. Brantley, Huiyu Feng, Daniel Hackenberg, Robert Henschel, Gabriele Jost, Daniel Molka, Chris Parrott, Joe Robichaux, and et al. SPEC OMP2012 – An Application Benchmark Suite for Parallel Systems Using OpenMP. *Lecture Notes in Computer Science (LNCS)*, 2012. DOI: 10.1007/978-3-642-30961-8_17.

[McC95] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995. online at computer.org (accessed 2016-10-22).

[MCS+13] Renato Miceli, Gilles Civario, Anna Sikora, Eduardo César, Michael Gerndt, Houssam Haitof, Carmen Navarrete, Siegfried Benkner, Martin Sandrieser, Laurent Morin, and Francois Bodin. AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications. In *Applied Parallel and Scientific Computing*, volume 7782 of *Lecture Notes in Computer Science (LNCS)*. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-36803-5_24.

[MDM+95] Shin'ichiro Mutoh, Takakuni Douseki, Yasuyuki Matsuya, Takahko Aoki, Satoshi Shigematsu, and Junzo Yamada. 1-V Power Supply High-Speed Digital Circuit Technology with Multithreshold-Voltage CMOS. *IEEE Journal of Solid-State Circuits*, 30(8), 1995. DOI: 10.1109/4.400426.

[Meu08] Hans Werner Meuer. TOP500 Project: Looking Back over 15 Years of Supercomputing Experience. *PIK-Praxis der Informationsverarbeitung und Kommunikation*, 31(2), 2008. DOI: 10.1515/piko.2008.0022.

[MHS14] Daniel Molka, Daniel Hackenberg, and Robert Schöne. Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer. *Workshop on Memory Systems Performance and Correctness (MSPC)*, 2014. DOI: 10.1145/2618128.2618129.

[MHSM09] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S. Müller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2009. DOI: 10.1109/PACT.2009.22.

[MHSM10] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S. Müller. Characterizing the Energy Consumption of Data Transfers and Arithmetic Operations on x86-64 Processors. *International Green Computing Conference (IGCC)*, Aug 2010. DOI: GREENCOMP.2010.5598316.

[MHSN15] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E. Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. *International Conference on Parallel Processing (ICPP)*, Sep 2015. DOI: 10.1109/ICPP.2015.83.

[Min13] Timo Minartz. *Design and Evaluation of Tool Extensions for Power Consumption Measurement in Parallel Systems*. PhD thesis, Universität Hamburg, 2013. online at uni-hamburg.de (accessed 2016-08-22).

[MKJ+07] Matthias S Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing Scalable Applications with Vampir, VampirServer and VampirTrace. *Advances in Parallel Computing*, 15: Parallel Computing: Architectures, Algorithms and Applications, 2007.

[MLPJ14] Abdelhafid Mazouz, Alexandre Laurent, Benoît Pradelle, and William Jalby. Evaluation of CPU Frequency Transition Latency. *Computer Science - Research and Development*, 29(3-4), 2014. DOI: 10.1007/s00450-013-0240-x.

[mpi15] MPI: A Message-Passing Interface Standard. Technical Report 3.0, Message Passing Interface Forum, Jun 2015. online at mpi-forum.org (accessed 2017-01-15).

[MPK06] Ananth Mavinakayanahalli, Prasanna Panchamukhi, and Jim Keniston. Probing the Guts of Kprobes. In *Proceedings of the Ottawa Linux Symposium (OLS)*, 2006. online at ibm.com (accessed 2016-10-20).

[MSHM11] Daniel Molka, Robert Schöne, Daniel Hackenberg, and Matthias S. Müller. Memory Performance and SPEC OpenMP Scalability on Quad-Socket x86_64 Systems. *Lecture Notes in Computer Science (LNCS)*, 2011. DOI: 10.1007/978-3-642-24650-0_15.

[MSHN17] Daniel Molka, Robert Schöne, Daniel Hackenberg, and Wolfgang E. Nagel. Detecting Memory-Boundedness with Hardware Performance Counters. In *Proceedings of the International Conference on Performance Engineering (ICPE) (accepted for publication)*, 2017.

[Mül13] Jupp Müller. Lastbalancierung von parallelen Anwendungen durch dynamische Taktfrequenzanpassung von Prozessoren, Diplomarbeit Technische Universität Dresden, 2013.

[MvWL+09] Matthias S. Müller, Matthijs van Waveren, Ron Lieberman, Brian Whitney, Hideki Saito, Kalyan Kumaran, John Baron, William C. Brantley, Chris Parrott, Tom Elken, Huiyu Feng, and Carl Ponder. SPEC MPI2007—an application benchmark suite for parallel systems using MPI. *Concurrency and Computation: Practice and Experience*, pages 191–205, 2009. DOI: 10.1002/cpe.v22:2.

[NMM+10] Aroon Nataraj, Allen D. Malony, Alan Morris, Dorian C. Arnold, and Barton P. Miller. A Framework for Scalable, Parallel Performance Monitoring. *Concurrenvy and Computation: Practice and Experience*, 22(6), 2010. DOI: 10.1002/cpe.1544.

[Ope15] OpenMP Architecture Review Board. OpenMP 4.5 specification, Nov 2015. online at openmp.org (accessed 2017-01-15).

[Ora14] Oracle Corporation. *Oracle Solaris Studio 12.4: OpenMP API User's Guide*, e37081 edition, Dec 2014. online at docs.oracle.com (accessed 2017-01-30).

[Pao10] Gabriele Paoloni. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*. Intel Corporation, 2010. online at intel.com (accessed 2017-01-02).

[Pat16] PathScale, LLC. *EKOPath User Guide*, 2016. online at pathscale.com (accessed 2016-06-12).

[PFJM10] Simone Pellegrini, Thomas Fahringer, Herbert Jordan, and Hans Moritsch. Automatic Tuning of MPI Runtime Parameter Settings by Using Machine Learning. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*. ACM, 2010. DOI: 10.1145/1787275.1787310.

[PKP03] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance. *Proceedings of the ACM/IEEE Conference on Supercomputing - SC 03*, 2003. DOI: 10.1145/1048935.1050204.

[PLB07] Venkatesh Pallipadi, Shaohua Li, and Adam Belay. cpuidle: Do nothing, efficiently. In *Proceedings of the Ottawa Linux Symposium (OLS)*, 2007. online at kernel.org (accessed 2017-01-12).

[PLCG95] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44. mar, 1995. online at citeseerx.ist.psu.edu (accessed 2017-01-31).

[PLR+13] Tapasya Patki, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. Exploring Hardware Overprovisioning in Power-constrained, High Performance Computing. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS)*, ICS '13, 2013. 10.1145/2464996.2465009.

[PLR+16] Tapasya Patki, David K. Lowenthal, Barry L. Rountree, Martin Schulz, and Bronis R. de Supinski. Economic Viability of Hardware Overprovisioning in Power-constrained High Performance Computing. In *Proceedings of the 4th International Workshop on Energy Efficient Supercomputing (E2SC)*, E2SC '16, 2016. DOI: 10.1109/E2SC.2016.12.

[Por16] The Portland Group, Inc. *PGI Compiler User's Guide for Intel 64 and AMD64 CPUs*, 2016 edition, May 2016. online at pgroup.com (accessed 2016-06-12).

[Pos] The open group base specifications. POSIX.1-2008 is simultaneously IEEE Std 1003.1-2008 and The Open Group Technical Standard Base Specifications, Issue 7, online at pubs.opengroup.org (accessed 2016-06-12).

[PPF12] S. Pellegrini, R. Prodan, and T. Fahringer. Tuning MPI runtime parameter setting for high performance computing. In *IEEE International Conference on Cluster Computing Workshops (CLUSTER WORKSHOPS)*, 2012. DOI: 10.1109/ClusterW.2012.15.

[PPH+13] Michael K. Patterson, Stephen W. Poole, Chung-Hsing Hsu, Don Maxwell, William Tschudi, Henry Coles, David J. Martinez, and Natalie Bates. TUE, a New Energy-Efficiency Metric Applied at ORNL's Jaguar. *Proceedings of the 28th International Conference on Supercomputing (ICS)*, 2013. DOI: 10.1007/978-3-642-38750-0_28.

[PS06] Venkatesh Pallipadi and Alexey Starikovskiy. The Ondemand Governor Past, Present, and Future. In *Proceedings of the Ottawa Linux Symposium (OLS)*, 2006. online at kernel.org (accessed 2017-01-12).

[QWPW97] CA Qing Wu, M. Pedram, and Xunwei Wu. Clock-Gating and Its Application to Low Power Design of Sequential Circuits. In *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC)*, 1997. DOI: 10.1109/CICC.1997.606671.

[Rei05] James Reinders. *VTune Performance Analyzer Essentials*. Intel Press, 2005. ISBN: 0974364959.

[RLdS+09] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. Adagio: Making DVS Practical for Complex HPC Applications. In *Proceedings of the 23rd International Conference on Supercomputing (ICS)*. ACM, 2009. DOI: 10.1145/1542275.1542340.

[RNA+12] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE International Symposium on Microarchitecture (MICRO)*, 32(2), Mar 2012. DOI: 10.1109/MM.2012.12.

[RNMM04] Efi Rotem, Alon Naveh, Micha Moffie, and Avi Mendelson. Analysis of Thermal Monitor Features of the Intel Pentium M Processor. In *Workshop on Temperature-Aware Computer Systems (TACS)*, 2004.

[Rob04] Stewart Robinson. *Simulation: The Practice of Model Development and Use*. John Wiley & Sons, 2004. ISBN: 0470847727.

[Rot11] Franz Rothlauf. *Design of Modern Heuristics*. Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-540-72962-4, ISBN: 3540729615.

[SBM09] Karan Singh, Major Bhadauria, and Sally A. McKee. Real Time Power Estimation and Thread Scheduling via Performance Counters. *ACM SIGARCH Computer Architecture News*, 37(2), 2009. DOI: 10.1145/1577129.1577137.

[Sch] SchedMD. *Slurm Power Saving Guide*. Slurm version 16.05, online at slurm.schedmd.com (accessed 2017-01-31).

[Sch11] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011. ISBN: 0982219199.

[SGK+17] Joseph Schuchart, Michael Gerndt, Per Gunnar Kjeldsberg, Michael Lysaght, David Horák, Lubomír Říha, Andreas Gocht, Mohammed Sourouri, Madhura Kumaraswamy, Anamika Chowdhury, Magnus Jahre, Kai Diethelm, Othman Bouizi, Umbreen Sabir Mian, Jakub Kružík, Radim Sojka, Martin Beseda, Venkatesh Kannan, Zakaria Bendifallah, Daniel Hackenberg, and Wolfgang E. Nagel. The READEX formalism for automatic tuning for energy efficiency. *Computing*, Jan 2017. DOI: 10.1007/s00607-016-0532-7.

[SGM+08] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. Open|SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis. *Scientific Programming*, 16(2-3), Apr 2008. DOI: 10.1155/2008/713705.

[SH11] Robert Schöne and Daniel Hackenberg. On-Line Analysis of Hardware Performance Events for Workload Characterization and Processor Frequency Scaling Decisions. In *Proceeding of the 2nd joint WOSP/SIPEW International Conference on Performance Engineering (ICPE)*. ACM, 2011. DOI: 10.1145/1958746.1958819.

[SHcF06] Sushant Sharma, Chung-Hsing Hsu, and Wu chun Feng. Making a case for a Green500 list. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium (IPDPS)*, April 2006. DOI: 10.1109/IPDPS.2006.1639600.

[SHM11] Robert Schöne, Daniel Hackenberg, and Daniel Molka. Simultaneous Multithreading on x86_64 Systems: An Energy Efficiency Evaluation. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems (HotPower)*. ACM, 2011. DOI: 10.1145/2039252.2039262.

[SHM12] Robert Schöne, Daniel Hackenberg, and Daniel Molka. Memory Performance at Reduced CPU Clock Speeds: An Analysis of Current x86_64 Processors. In *Proceedings of the 5th Workshop on Power-Aware Computing and Systems (HotPower)*. USENIX Association, 2012. online at usenix.org (accessed 2017-01-31).

[SIB+16] Robert Schöne, Thomas Ilsche, Mario Bielert, Daniel Molka, and Daniel Hackenberg. Software Controlled Clock Modulation for Energy Efficiency Optimization on Intel Processors. In *Proceedings of the 4th International Workshop on Energy Efficient Supercomputing (E2SC)*, E2SC '16. IEEE Press, 2016. DOI: 10.1109/E2SC.2016.15.

[SKK11] Vasileios Spiliopoulos, Stefanos Kaxiras, and Georgios Keramidas. Green Governors: A Framework for Continuously Adaptive DVFS. In *International Green Computing Conference and Workshops (IGCC)*. IEEE, 2011. DOI: 10.1109/IGCC.2011.6008552.

[SKM13] Robert Schöne, Andreas Knüpfer, and Daniel Molka. Potentials and Limitations for Energy Efficiency Auto-Tuning. In *Advances in Parallel Computing*, volume Volume 25: Parallel Computing: Accelerating Computational Science and Engineering (CSE), 2013. DOI: 10.3233/978-1-61499-381-0-678.

[SKVM15] Pavel Saviankou, Michael Knobloch, A. Visser, and Bernd Mohr. Cube v4: From Performance Report Explorer to Performance Analysis Tool. *Procedia Computer Science*, 51, June 2015. DOI: 10.1016/j.procs.2015.05.320.

[SM06] Sameer S. Shende and Allen D. Malony. The TAU Parallel Performance System. *The International Journal of High Performance Computing Applications (IJHPCA)*, 20(2), 2006. DOI: 10.1177/1094342006064482.

[SM13] Robert Schöne and Daniel Molka. Integrating Performance Analysis and Energy Efficiency Optimizations in a Unified Environment. *Computer Science - Research and Development*, 2013. DOI: 10.1007/s00450-013-0243-7.

[SMW14] Robert Schöne, Daniel Molka, and Michael Werner. Wake-up Latencies for Processor Idle States on Current x86 Processors. *Computer Science - Research and Development*, 2014. DOI: 10.1007/s00450-014-0270-z.

[SSIH14] Robert Schöne, Joseph Schuchart, Thomas Ilsche, and Daniel Hackenberg. Scalable Tools for Non-Intrusive Performance Debugging of Parallel Linux Workloads. In *Proceedings of*

the Ottawa Linux Symposium (OLS), 2014. online at http://kernel.org (accessed 2017-01-10).

[STD+14] Robert Schöne, Jan Treibig, Manuel F. Dolz, Carla Guillen, Carmen Navarrete, Michael Knobloch, and Barry Rountree. Tools and Methods for Measuring and Tuning the Energy Efficiency of HPC Systems. *Scientific Programming*, 22(4), 2014. DOI: 10.3233/SPR-140393.

[STHI11] Robert Schöne, Ronny Tschüter, Daniel Hackenberg, and Thomas Ilsche. The VampirTrace Plugin Counter Interface: Introduction and Examples. In *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2011. DOI: 10.1007/978-3-642-21878-1_62.

[STHW15] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS)*, 2015. DOI: 10.1145/2751205.2751240.

[STI+17] Robert Schöne, Ronny Tschüter, Thomas Ilsche, Joseph Schuchart, Daniel Hackenberg, and Wolfgang E. Nagel. Extending the Functionality of Score-P through Plugins: Interfaces and Use Cases. In *Tools for High Performance Computing 2016 (accepted for publication)*. Springer, 2017.

[Tay12] Michael B. Taylor. Is Dark Silicon Useful?: Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Proceedings of the 49th Annual Design Automation Conference (DAC)*, 2012. 10.1145/2228360.2228567.

[TH10] Jan Treibig and Georg Hager. Introducing a Performance Model for Bandwidth-Limited Loop Kernels. *Lecture Notes in Computer Science (LNCS)*, 2010. DOI: 10.1007/978-3-642-14390-8_64.

[TLP+12] Anish Tiwari, Michael Laurenzano, Joshua Peraza, Laura Carrington, and Allan Snavely. Green queue: Customized Large-Scale Clock Frequency Scaling. In *International Conference on Cloud and Green Computing (CGC)*. IEEE, 2012. DOI: 10.1109/CGC.2012.62.

[Vet15] Jeffrey S. Vetter. *Contemporary High Performance Computing: From Petascale toward Exascale, Volume Two*. Chapman & Hall/CRC Computational Science. CRC Press, 3 2015. ISBN: 1466568348.

[Wat] Watts up? *Serial Data Format Revision: 4.4.1*. online at wattsupmeters.com (accessed 2017-01-31).

[Wat09] Philip Watson. *ARM: 1176 IEM Reference Methodology*. ARM, jun 2009. online at si2.org (accessed 2017-01-15).

[WD98] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC)*. IEEE Computer Society, 1998. online at netlib.org (accessed 2017-01-30).

[WDF+14] Jons-Tobias Wamhoff, Stephan Diestelhorst, Christof Fetzer, Patrick Marlier, Pascal Felber, and Dave Dice. The TURBO Diaries: Application-controlled Frequency Scaling Explained. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014. online at usenix.org (accessed 2017-01-31).

[Wea13] Vincent M Weaver. Linux perf_event Features and Overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, 2013.

[Wea15] Vincent M. Weaver. Self-monitoring Overhead of the Linux perf_event Performance Counter Interface. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar 2015. DOI: 10.1109/ISPASS.2015.7095789.

[Wes11] Weste, Neil H. E. and Harris, David M. *CMOS VLSI Design - A Circuits and Systems Perspective, 4th Edition*. Pearson, 2011. ISBN: 0321547748.

[WJK+12] Vincent M Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. Measuring energy and power with PAPI. In *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW)*. IEEE, 2012. DOI: 10.1109/ICPPW.2012.39.

[WM16] Scott Walker and Marty McFadden. Best Practices for Scalable Power Measurement and Control. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016. DOI: 10.1109/IPDPSW.2016.9.

[WMF+15] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned Global Address Space Languages. *ACM Computing Surveys (CSUR)*, 47(4), 2015. DOI: 10.1145/2716320.

[WRF+10] Malcolm Ware, Karthick Rajamani, Michael Floyd, Bishop Brock, Juan C Rubio, Freeman Rawson, and John B Carter. Architecting for Power Management: The IBM POWER7 Approach. In *IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2010. DOI: 10.1109/HPCA.2010.5416627.

[WSM15] Bo Wang, Dirk Schmidl, and Matthias S. Müller. Evaluating the Energy Consumption of OpenMP Applications on Haswell Processors. *Lecture Notes in Computer Science (LNCS)*, 2015. DOI: 10.1007/978-3-319-24595-9_17.

[WT15] Xingfu Wu and Valerie Taylor. Power and performance characteristics of CORAL Scalable Science Benchmarks on BlueGene/Q Mira. *Sixth International Green and Sustainable Computing Conference (IGSC)*, Dec 2015. DOI: 10.1109/IGCC.2015.7393681.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM (CACM)*, 52(4), 2009. DOI: 10.1145/1498765.1498785.

[ZES11] ZES ZIMMER Electronic Systems GmbH. *4 Channel Power Meter LMG450 User manual*, 2011.

[ZFD+15] V. Zyuban, J. Friedrich, D. M. Dreps, J. Pille, D. W. Plass, P. J. Restle, Z. T. Deniz, M. M. Ziegler, S. Chu, S. Islam, and et al. IBM power8 circuit design and energy optimization. *IBM Journal Research & Development*, 59(1), Jan 2015. DOI: 10.1147/JRD.2014.2380200.

List of Figures

List of Tables

A Test Systems

So computers are tools of the devil? thought Newt. He had no problem believing it. Computers had to be the tools of somebody, and all he knew for certain was that it definitely wasn't him.
Good Omens by Neil Gaiman & Terry Pratchett

To gather the section for this data, I used a shell script that calls `hwloc` and `lstopo`. Thus, most of it is created automatically.

A.1 Overview

	Architecture	Model	Frequency Range [GHz]	Uncore Frequency Range [GHz]	Vendor Sources
Desktop	Sandy Bridge	i7-2600	1.6-3.4	-	[Int11a, Int13b, Int11b]
	Ivy Bridge	i5-3470	1.6-3.2	-	[Int13c, Int13d, Int13e]
	Haswell	i7-4770	0.8-3.4	0.8-3.9	[Int13f, Int14a, Int14b]
	Skylake	i7-6700K	0.5-4.0	0.8-4.1	[Int16c, Int16d]
	Kaveri	A10-7850K	1.7-3.7	1.8	[Adv15]
Server	Westmere	Xeon X5670	1.6-2.93	2.67	[Int11d, Int10]
	Sandy Bridge	Xeon E5-2670	1.2-2.6	-	[Int12b, Int12c]
	Haswell	Xeon E5-2680 v3	1.2-2.6	1.2-3.0	[Int15b, Int15c]
	Bulldozer	Opteron 6274	1.4-2.2	2.0	[Adv13]

A.2 Desktop Processors

A.2.1 Intel Core i7-2600

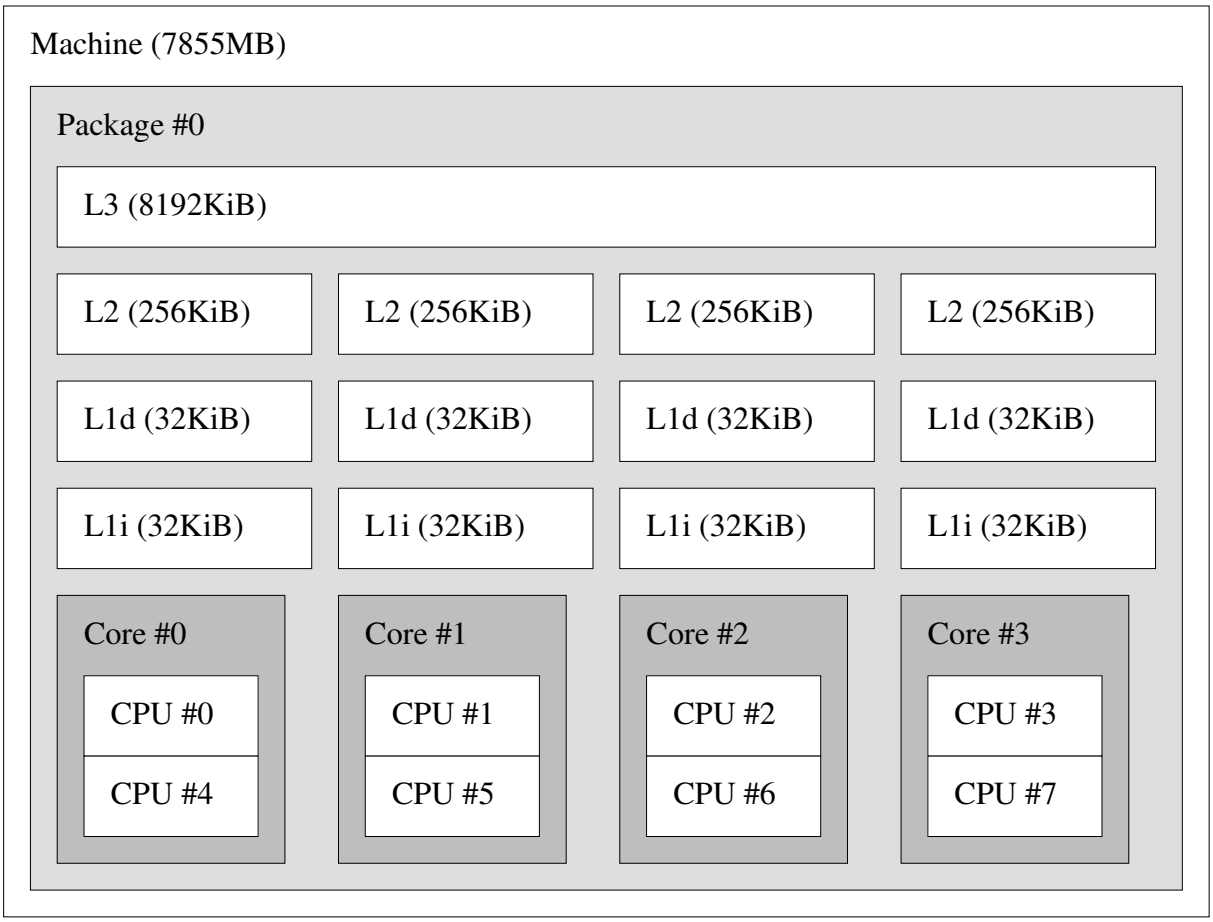


Figure A.1: Topology information for Intel Core i7-2600 (gathered with `lstopo`)

Location	Size	Vendor	Part Number
DIMM-1	4 GiB	Samsung	M378B5273DH0-CH9
DIMM-2	4 GiB	Samsung	M378B5273DH0-CH9

Table A.2: Memory set-up of Intel Core i7-2600 test system, total 8 GiB (gathered with `hwinfo`)

System Vendor	FUJITSU
Product Description	ESPRIMO P700
Mainboard Vendor	FUJITSU
Mainboard Description	D3061-A1
Memory Size	8 GiB

Table A.3: System description of Intel Core i7-2600 test system (gathered with `hwinfo`)

A.2.2 Intel Core i5-3470

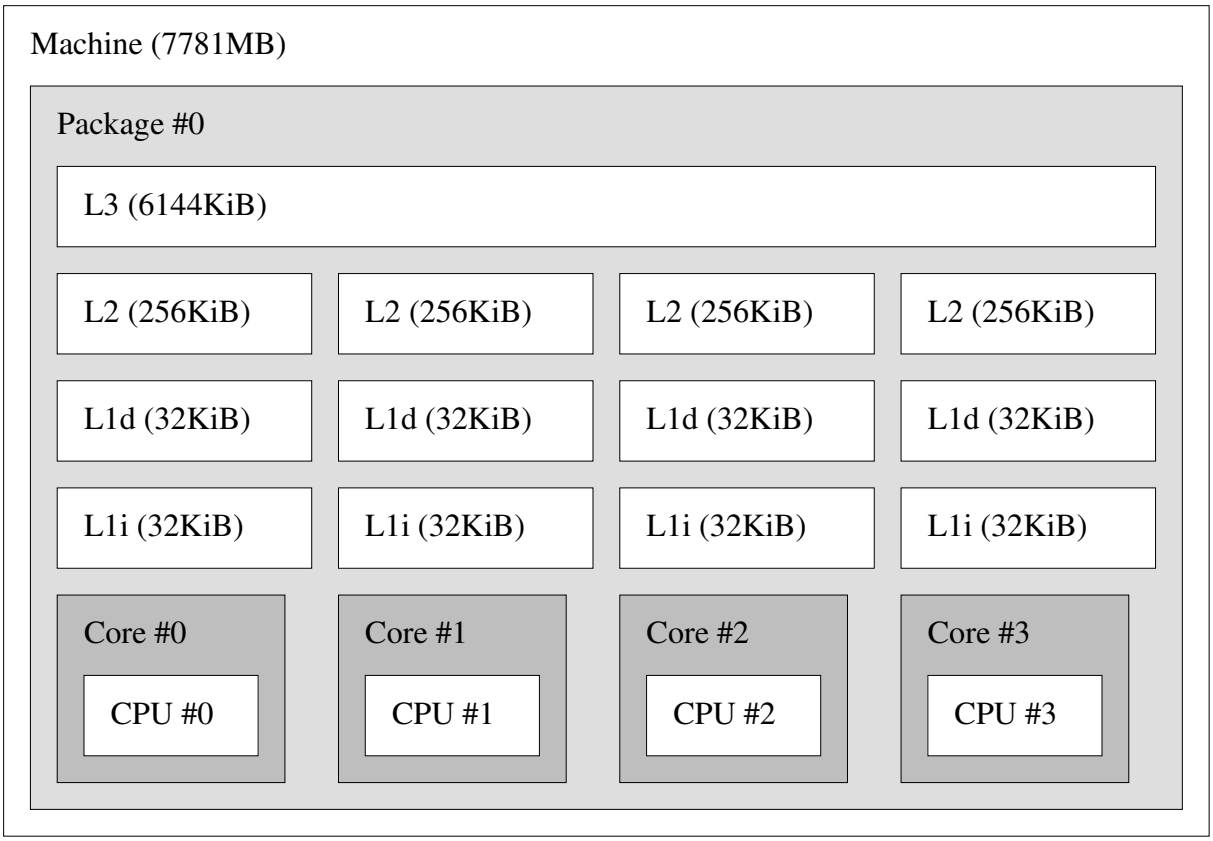


Figure A.2: Topology information for Intel Core i5-3470 (gathered with `lstopo`)

Location	Size	Vendor	Part Number
DIMM A1	4 GiB	Samsung	M378B5273DH0-CK0
DIMM B2	4 GiB	Samsung	M378B5273DH0-CK0

Table A.4: Memory set-up of Intel Core i5-3470 test system, total 8 GiB (gathered with `hwinfo`)

System Vendor	FUJITSU
Product Description	ESPRIMO P910
Mainboard Vendor	FUJITSU
Mainboard Description	D3162-A1
Memory Size	8 GiB

Table A.5: System description of Intel Core i5-3470 test system (gathered with `hwinfo`)

A.2.3 Intel Core i7-4770

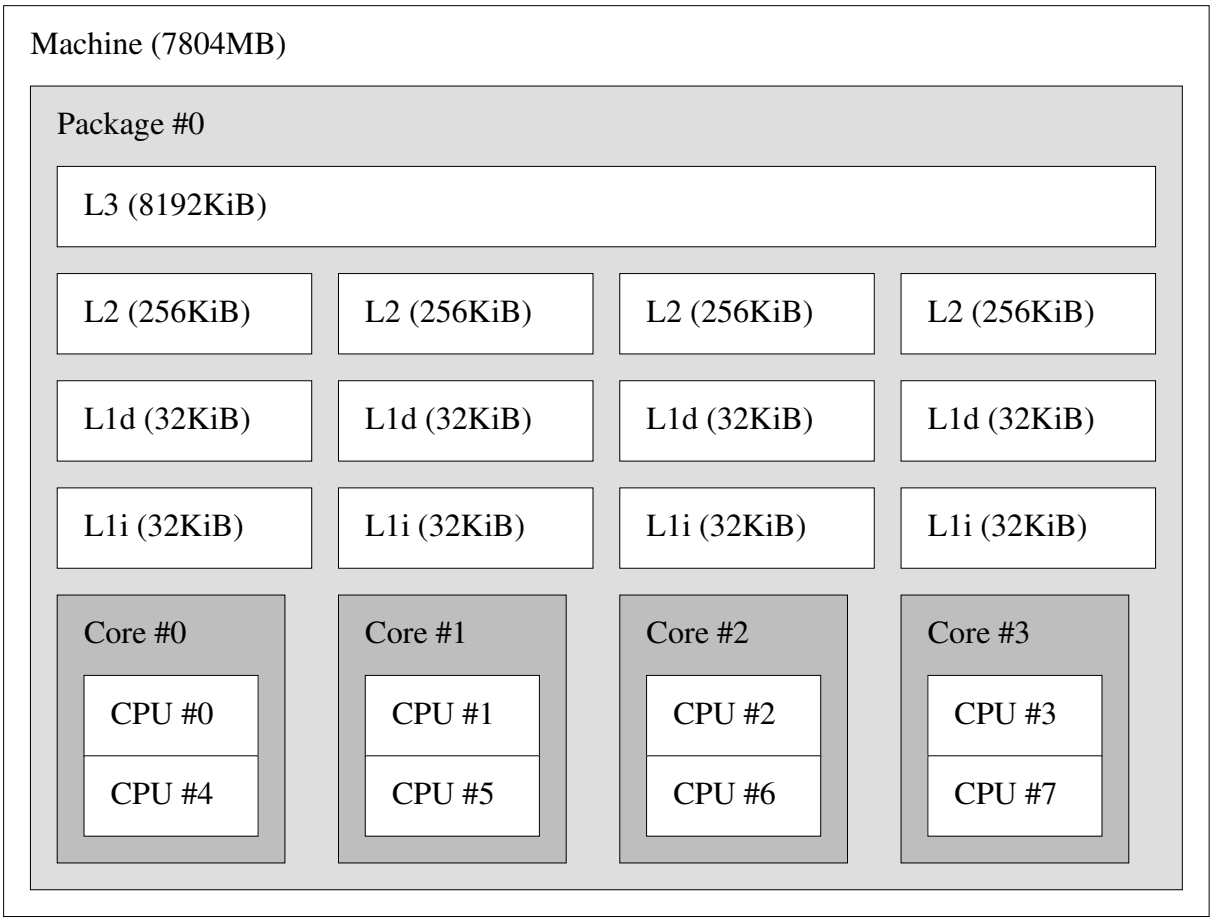


Figure A.3: Topology information for Intel Core i7-4770 (gathered with `lstopo`)

Location	Size	Vendor	Part Number
DIMM A1	4 GiB	Samsung	M378B5273DH0-CK0
DIMM B2	4 GiB	Samsung	M378B5273DH0-CK0

Table A.6: Memory set-up of Intel Core i7-4770 test system, total 8 GiB (gathered with `hwinfo`)

System Vendor	FUJITSU
Product Description	ESPRIMO P920
Mainboard Vendor	FUJITSU
Mainboard Description	D3222-A1
Memory Size	8 GiB

Table A.7: System description of Intel Core i7-4770 test system (gathered with `hwinfo`)

A.2.4 Intel Core i7-6700K

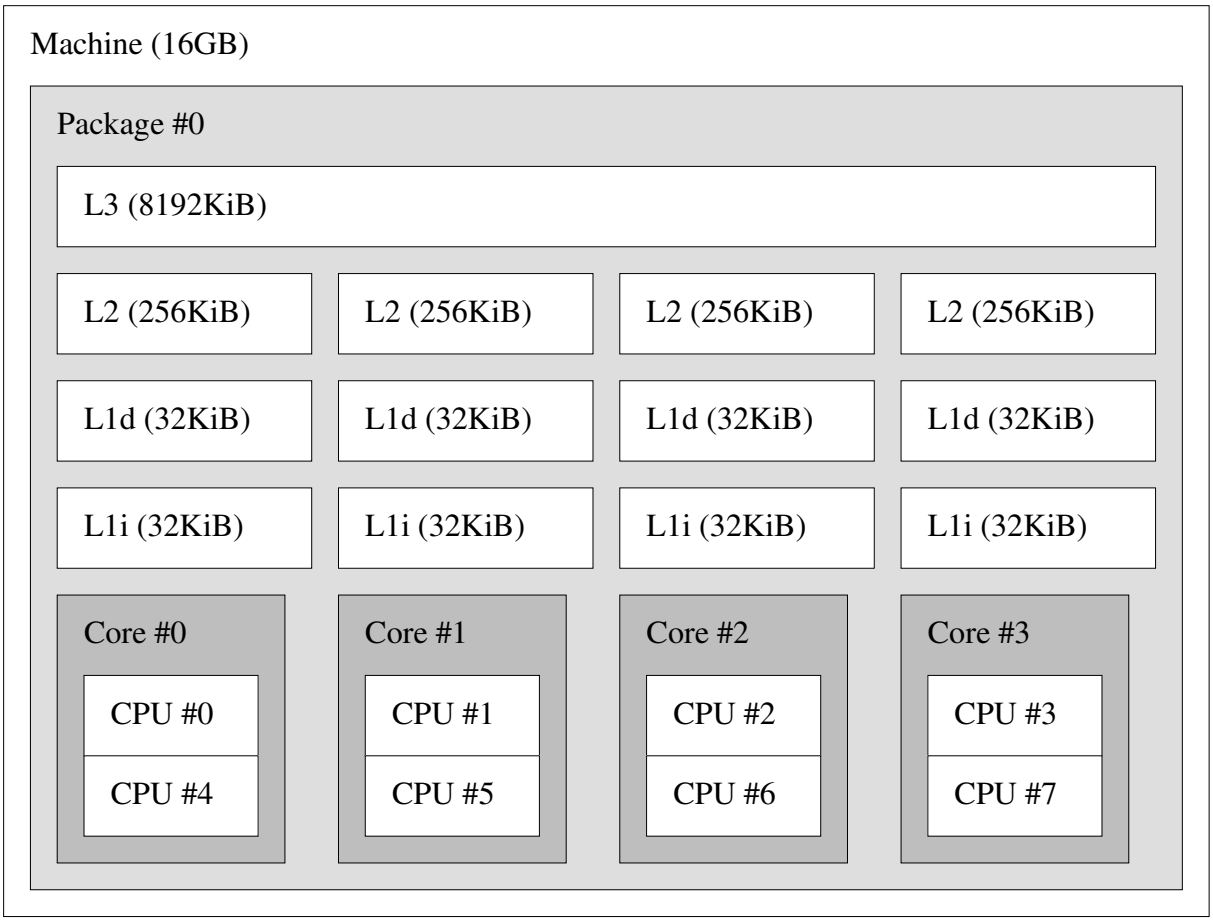


Figure A.4: Topology information for Intel Core i7-6700K (gathered with `lstopo`)

Location	Size	Vendor	Part Number
DIMM-A1	8 GiB	CRUCIAL	CT8G4DFS8213.M8FA
DIMM-B1	8 GiB	CRUCIAL	CT8G4DFS8213.M8FA

Table A.8: Memory set-up of Intel Core i7-6700K test system, total 16 GiB (gathered with `hwinfo`)

System Vendor	TAROX
Product Description	Basic PC System
Mainboard Vendor	ASUSTeK COMPUTER INC.
Mainboard Description	Z170M-PLUS
Memory Size	16 GiB

Table A.9: System description of Intel Core i7-6700K test system (gathered with `hwinfo`)

A.2.5 AMD A10-7850K

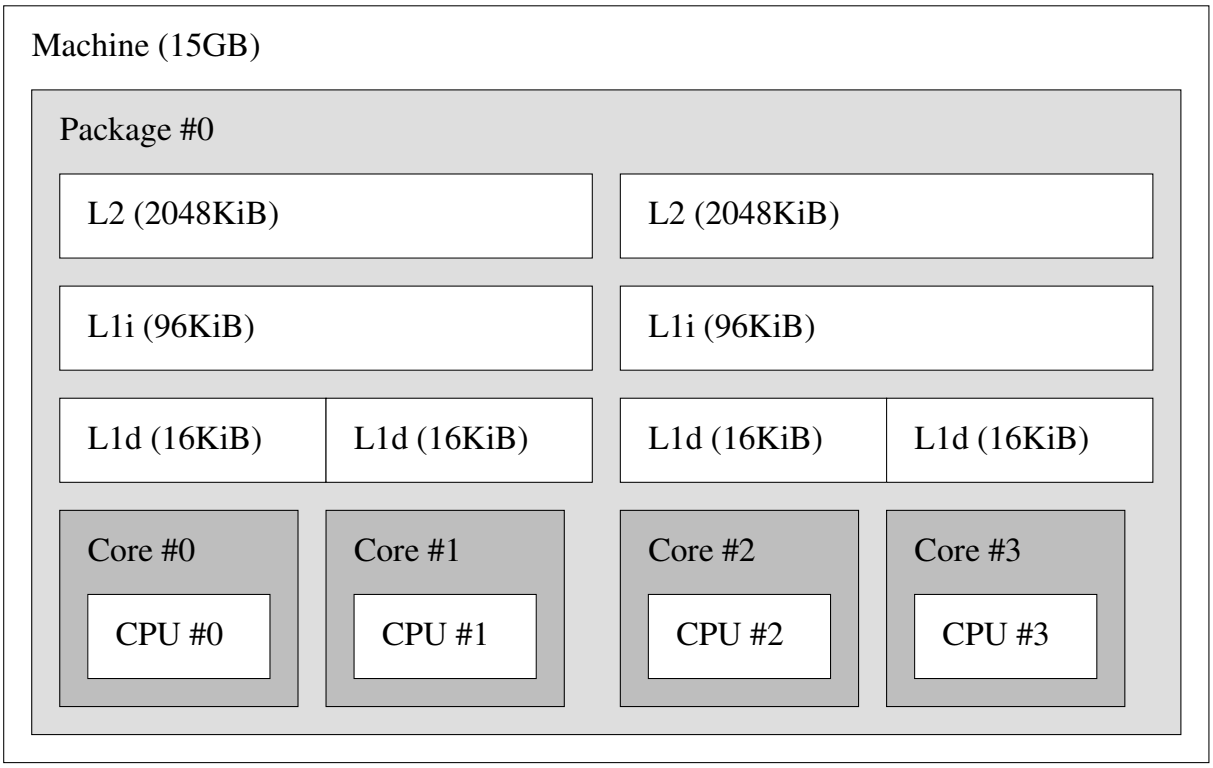


Figure A.5: Topology information for AMD A10-7850K (gathered with `lstopo`)

Location	Size	Vendor	Part Number
DIMM-A2	8 GiB	Undefined	F3-2133C9-8GXH
DIMM-B2	8 GiB	A1-Manufacturer3	Array1-PartNumber3

Table A.10: Memory set-up of AMD A10-7850K test system, total 16 GiB (gathered with `hwinfo`)

System Vendor	System manufacturer
Product Description	System Product Name
Mainboard Vendor	ASUSTeK COMPUTER INC.
Mainboard Description	A88X-PRO
Memory Size	16 GiB

Table A.11: System description of AMD A10-7850K test system (gathered with `hwinfo`)

A.3 Server Processors

A.3.1 Intel Xeon X5670

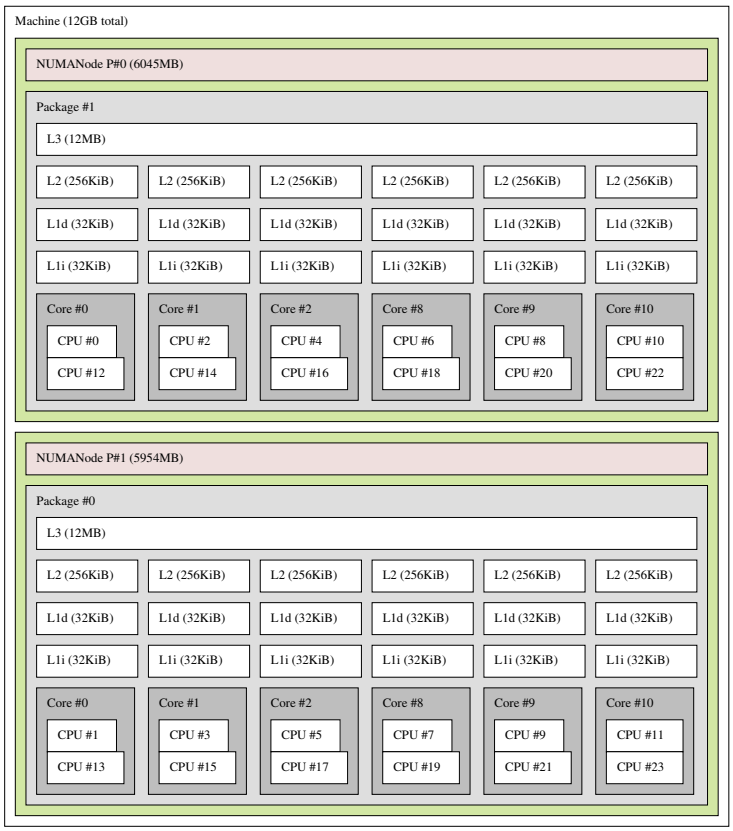


Figure A.6: Topology information for Intel Xeon X5670 (gathered with `lstopo`)

Location	Size	Vendor	Part Number
DIMM-A1	2 GiB	00CE00B380CE	M393B5773CH0-YH9
DIMM-A2	2 GiB	00CE00B380CE	M393B5773CH0-YH9
DIMM-A3	2 GiB	00CE00B380CE	M393B5773CH0-YH9
DIMM-B1	2 GiB	00CE00B380CE	M393B5773CH0-YH9
DIMM-B2	2 GiB	00CE00B380CE	M393B5773CH0-YH9
DIMM-B3	2 GiB	00CE00B380CE	M393B5773CH0-YH9

Table A.12: Memory set-up of Intel Xeon X5670 test system, total 12 GiB (gathered with `hwinfo`)

System Vendor	Dell Inc.
Product Description	PowerEdge R510
Mainboard Vendor	Dell Inc.
Mainboard Description	0DPRKF
Memory Size	12 GiB

Table A.13: System description of Intel Xeon X5670 test system (gathered with `hwinfo`)

A.3.2 Intel Xeon E5-2670

Figure A.7: Topology information for Intel Xeon E5-2670 (gathered with `lstopo`)

Location	Size	Vendor	Part Number
DIMM-A1	8 GiB	00AD00B300AD	HMT31GR7CFR4C-PB
DIMM-A2	8 GiB	00AD00B300AD	HMT31GR7CFR4C-PB
DIMM-A3	8 GiB	00AD00B300AD	HMT31GR7CFR4C-PB
DIMM-A4	8 GiB	00AD00B300AD	HMT31GR7CFR4C-PB
DIMM-B1	8 GiB	00AD00B300AD	HMT31GR7CFR4C-PB
DIMM-B2	8 GiB	00AD00B300AD	HMT31GR7CFR4C-PB
DIMM-B3	8 GiB	00AD00B300AD	HMT31GR7CFR4C-PB
DIMM-B4	8 GiB	00AD00B300AD	HMT31GR7CFR4C-PB

Table A.14: Memory set-up of Intel Xeon E5-2670 test system, total 64 GiB (gathered with `hwinfo`)

System Vendor	Dell Inc.
Product Description	PowerEdge R720
Mainboard Vendor	Dell Inc.
Mainboard Description	0M1GCR
Memory Size	64 GiB

Table A.15: System description of Intel Xeon E5-2670 test system (gathered with `hwinfo`)

A.3.3 Intel Xeon E5-2680 v3

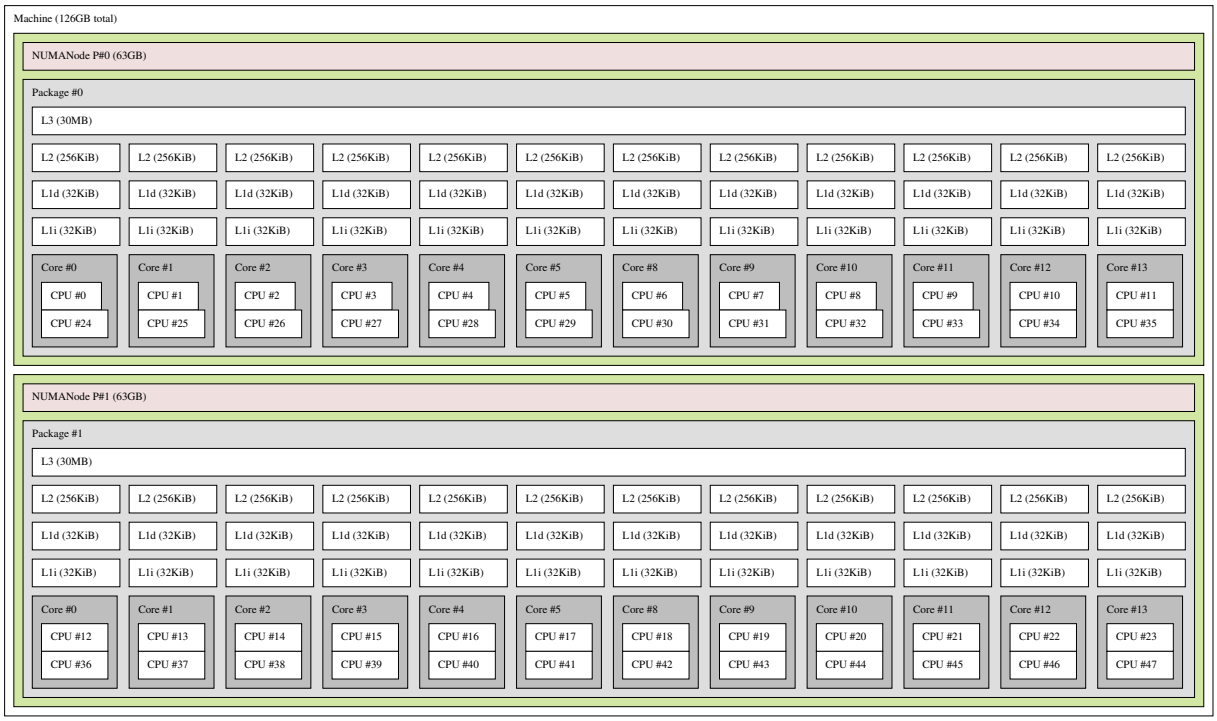


Figure A.8: Topology information for Intel Xeon E5-2680 v3 (gathered with `lstopo`)

Location	Size	Vendor	Part Number
P1-DIMMC1	16 GiB	Micron(data:14/21)	36ASF2G72PZ-2G1A2
P1-DIMMD1	16 GiB	Micron(data:14/21)	36ASF2G72PZ-2G1A2
P2-DIMME1	16 GiB	Micron(data:14/21)	36ASF2G72PZ-2G1A2
P2-DIMMF1	16 GiB	Micron(data:14/21)	36ASF2G72PZ-2G1A2
P2-DIMMG1	16 GiB	Micron(data:14/21)	36ASF2G72PZ-2G1A2
P2-DIMMH1	16 GiB	Micron(data:14/21)	36ASF2G72PZ-2G1A2
P1-DIMMA1	16 GiB	Micron(data:14/21)	36ASF2G72PZ-2G1A2
P1-DIMMB1	16 GiB	Micron(data:14/21)	36ASF2G72PZ-2G1A2

Table A.16: Memory set-up of Intel Xeon E5-2680 v3 test system, total 128 GiB (gathered with `hwinfo`)

System Vendor	bullx
Product Description	R421-E4
Mainboard Vendor	Supermicro
Mainboard Description	X10DRG-H
Memory Size	128 GiB

Table A.17: System description of Intel Xeon E5-2680 v3 test system (gathered with `hwinfo`)

A.3.4 AMD Opteron 6274

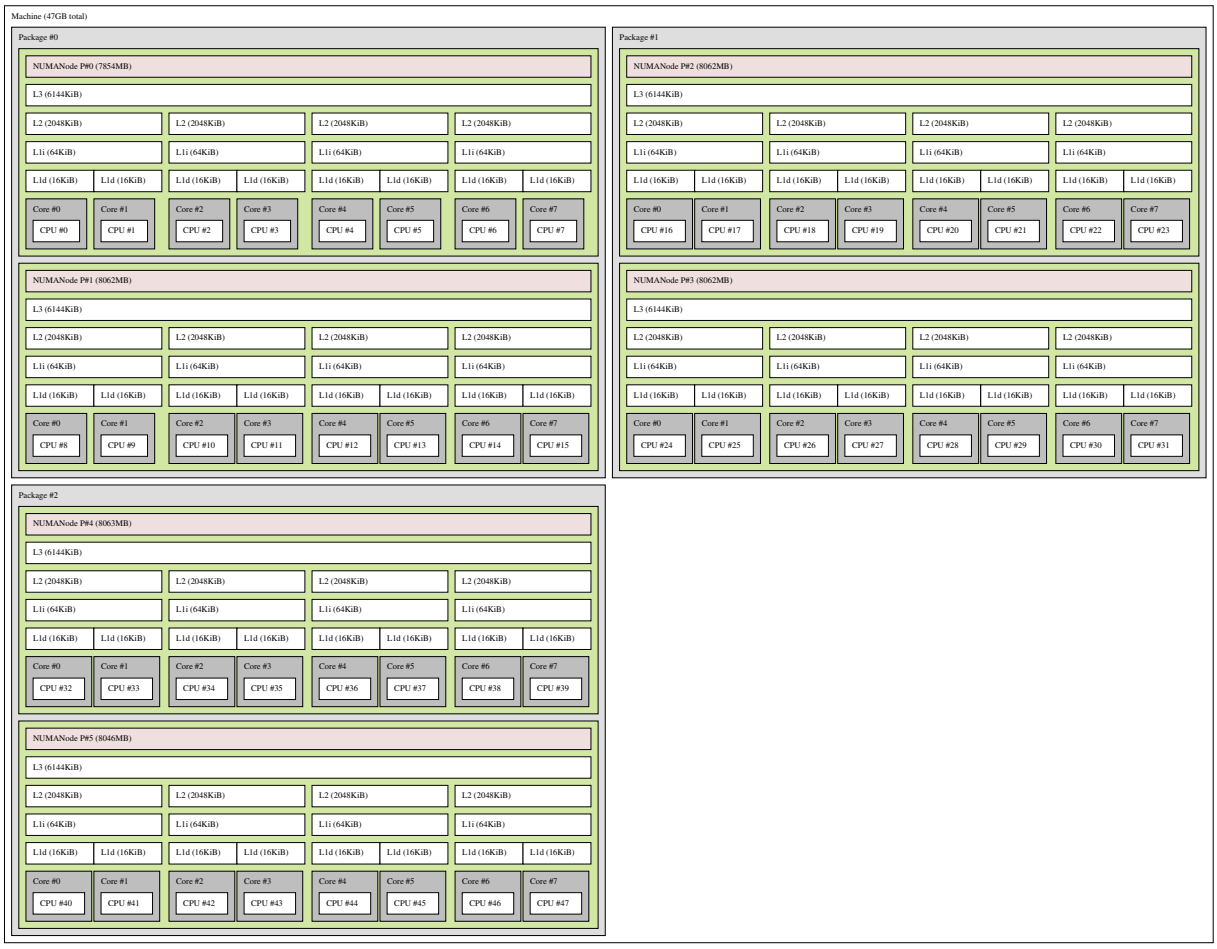


Figure A.9: Topology information for AMD Opteron 6274 (gathered with `lstopo`)

Location	Size	Vendor	Part Number
P1-1A	4 GiB	Samsung	M393B5270DH0-CK0
P1-2A	4 GiB	Samsung	M393B5270DH0-CK0
P1-3A	4 GiB	Samsung	M393B5270DH0-CK0
P3-1A	4 GiB	Samsung	M393B5270DH0-CK0
P3-2A	4 GiB	Samsung	M393B5270DH0-CK0
P3-3A	4 GiB	Samsung	M393B5270DH0-CK0
P3-4A	4 GiB	Samsung	M393B5270DH0-CK0
P4-1A	4 GiB	Samsung	M393B5270DH0-CK0
P4-2A	4 GiB	Samsung	M393B5270DH0-CK0
P4-3A	4 GiB	Samsung	M393B5270DH0-CK0
P4-4A	4 GiB	Samsung	M393B5270DH0-CK0

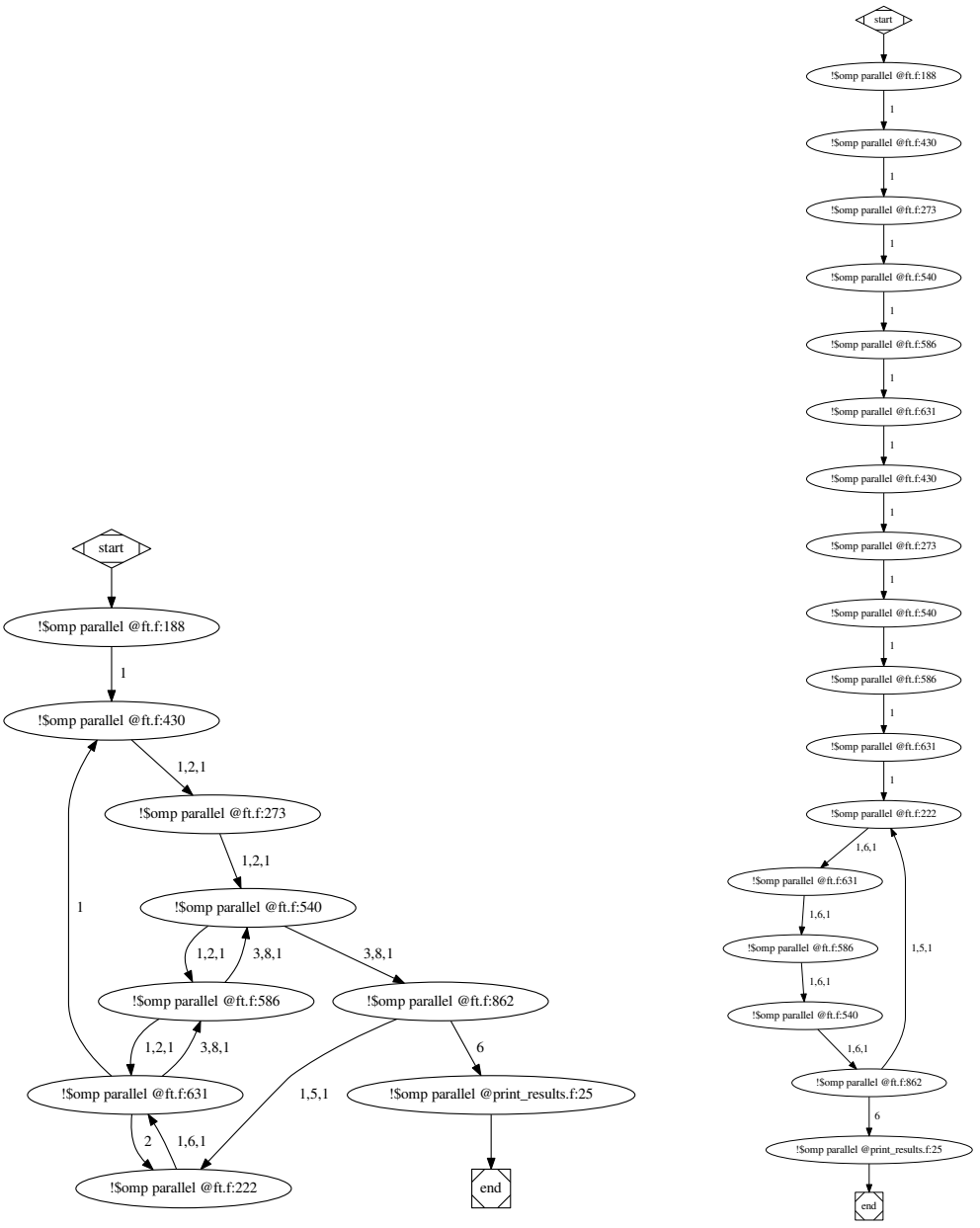
Table A.18: Memory set-up of AMD Opteron 6274 test system, total 44 GiB (gathered with `hwinfo`)

System Vendor	Supermicro
Product Description	H8QGL
Mainboard Vendor	Supermicro
Mainboard Description	H8QGL
Memory Size	44 GiB

Table A.19: System description of AMD Opteron 6274 test system (gathered with `hwinfo`)

B Supplemental Figures

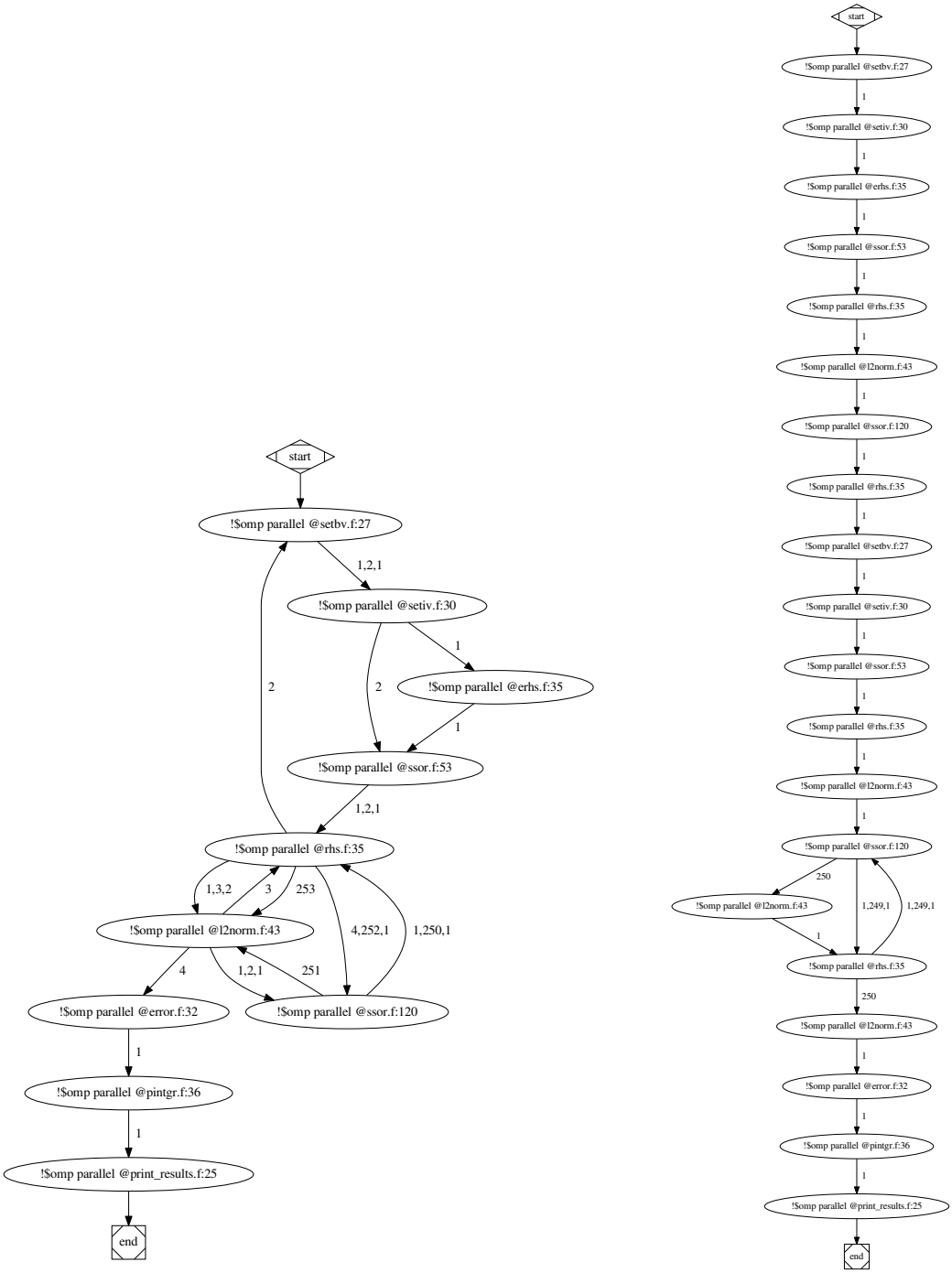
"And you would know all this if you had troubled to read the book that Maester Kedry gave you."
"It had no pictures." Dialogue between Quentyn Martell and Gerris Drinkwater
A Dance with Dragons by George Raymond Richard Martin



(a) Context free

(b) With context

Figure B.1: EFG of NPB FT (size A). Nodes represent parallel regions, edges depict transitions.



(a) Context free

(b) With context

Figure B.2: EFG of NPB LU (size A). Nodes represent parallel regions, edges depict transitions.

C Supplemental Algorithms

Mathematical analysis and computer modeling are revealing to us that the shapes and processes we encounter in nature—the way that plants grow, the way that mountains erode or rivers flow, the way that snowflakes or islands achieve their shapes, the way that light plays on a surface, the way the milk folds and spins into your coffee as you stir it, the way that laughter sweeps through a crowd of people—all these things in their seemingly magical complexity can be described by the interaction of mathematical processes that are, if anything, even more magical in their simplicity. , Richard McDuff
Dirk Gently's Holistic Detective Agency by Douglas Adams

Algorithm C.1 Performance measurement in STREAM benchmark [McC95]

```
function STREAM(NTIMES,STREAM_ARRAY_SIZE)
    ...                                              ▷ Initialize, sets a,b,c,scalar,times
    for k=0;k<NTIMES;k++ do
        times[0][k] ← gettime()
        for j=0;j<STREAM_ARRAY_SIZE;j++ do           ▷ Copy, OpenMP parallel
            c[j] ← a[j]
        end for
        times[0][k] ← gettime()-times[0][k]
        times[1][k] ← gettime()
        for j=0;j<STREAM_ARRAY_SIZE;j++ do           ▷ Scale, OpenMP parallel
            b[j] ← scalar * c[j]
        end for
        times[1][k] ← gettime()-times[1][k]
        times[2][k] ← gettime()
        for j=0;j<STREAM_ARRAY_SIZE;j++ do           ▷ Add, OpenMP parallel
            c[j] ← a[j] + b[j]
        end for
        times[2][k] ← gettime()-times[2][k]
        times[3][k] ← gettime()
        for j=0;j<STREAM_ARRAY_SIZE;j++ do           ▷ Triad, OpenMP parallel
            a[j] ← b[j] + scalar * c[j]
        end for
        times[3][k] ← gettime()-times[3][k]
    end for
end function
```

Algorithm C.2 Version to measure Haswell-EP transition latencies, italic: code that has been introduced

```
function MEASURE(target_time_interval, target_frequency, source_frequency, wait_time,
nr_measurements)
    for i=0; i<nr_measurements;i+=1 do
        cpufreq_setspeed(cpu0,source_frequency)
        repeat
            cpufreq ← measure_cpufreq()
        until cpufreq ← source_frequency
        wait(wait_time)
        cpufreq_setspeed(cpu0,target_frequency)
        repeat
            start_time ← rdtsc()
            measurement_loop()
            stop_time ← rdtsc()
        until stop_time - start_time ∈ target_time_interval
        ...                                                    ▷ Verify measurement
        if verified then
            measured_time[i] ← stop_time - init_time
        else
            i-=1;
        end if
    end for
end function
```

Algorithm C.3 Load balancing substrate algorithm

function INITIALIZE THREAD
  $targets \leftarrow \emptyset$ ▷ Initially, there are no known functions
  $target \leftarrow None$ ▷ There is no function that has been executed before
end function
function ENTER SYNCH. FUNCTION($metrics$) ▷ Called when synchronizing region is entered
  $time_{\text{compute end}} \leftarrow metrics.time$ ▷ Gather time ...
  $cycles_{\text{compute end}} \leftarrow metrics.cycles$ ▷ ... and cycles to determine frequency and ...
  ▷ ... computation time for current $target$ when the function is exited
end function
function EXIT SYNCH. FUNCTION($metrics$) ▷ Called when synchronizing region is exited
  if $target \neq None$ **then** ▷ Not for the first exit call, where f_{last} is not yet set
    $time_{\text{synchronization end}} \leftarrow metrics.time$
    $cycles_{\text{synchronization end}} \leftarrow metrics.cycles$
    $time_{compute} \leftarrow time_{\text{compute end}} - time_{\text{compute start}}$ ▷ Get computation time for last function
    $f_{measured} \leftarrow \frac{cycles_{\text{compute end}} - cycles_{\text{compute start}}}{time_{compute}}$ ▷ Get frequency for last function
    $time_{sync} \leftarrow time_{\text{synchronization end}} - time_{\text{compute end}}$ ▷ Synchronization time for last function
    if $time_{compute} + time_{sync} > t_{total_{min}}$ **then** ▷ If the region is long enough to optimize
      if $\frac{time_{compute}}{time_{compute}+time_{sync}} > 0.95 \vee time_{sync} < t_{sync_{min}}$ **then** ▷ If more than 95 % compute
        $f_{target} \leftarrow f_{reference}$ ▷ increase frequency ...
      else if $\frac{time_{compute}}{time_{compute}+time_{sync}} > 0.85$ **then** ▷ ..., otherwise if close to barrier...
        $f_{target} \leftarrow f_{measured}$ ▷ ... keep frequency ...
      else ▷ ..., otherwise if far from barrier ...
        $f_{target} \leftarrow f_{measured} * \frac{time_{compute}}{time_{compute}+time_{sync}} + 200MHz$ ▷ ... reduce frequency.
      end if
      $f_{target} \leftarrow \min(f_{target}, f_{ref})$ ▷ Avoid going over reference frequency
      $target.pop()$ ▷ Remove oldest target frequency
      $target.push(f_{target})$ ▷ Add new target frequency
    else
      $target.pop()$ ▷ Remove oldest target frequency
      $target.push(\infty)$ ▷ Mark to ignore this region for optimization
    end if
  end if
  $target \leftarrow targets[metrics.stack_{id}]$ ▷ Get statistics for follow up functions
  if $target == None$ **then** ▷ If there is no function for the current stack, create one
    $target \leftarrow [f_{ref}, f_{ref}, f_{ref}, f_{ref}]$ ▷ Initial target frequency is reference frequency
    $targets[metrics.stack_{id}] \leftarrow target$ ▷ Store buffer for further reference
  end if
  if $\max(target) \neq \infty$ **then** ▷ If this region is not to be ignored
    set_frequency($\max(target)$) ▷ Set frequency based on max. from previous target frequencies
  end if
  $time_{\text{compute start}} \leftarrow metrics.time$ ▷ Measure new time ...
  $cycles_{\text{compute start}} \leftarrow metrics.cycles$ ▷ ... and frequency for current $target$
end function

D Supplemental Listings

If you've never programmed a computer, you should. There's nothing like it in the whole world. When you program a computer, it does exactly what you tell it to do., Marcus Yallow
Little Brother, by Cory Doctorow

Marcus Yallow

```
#include <vampirtrace/vt_plugin_cntr.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <linux/perf_event.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/ioctl.h>
#include <pthread.h>


static int nr=0;

static pthread_mutex_t add_counter_mutex;

int32_t init(){
  /* check if pthread mutex can be created */
  return pthread_mutex_init( &add_counter_mutex, NULL );
}


int32_t add_counter(char * event_name){
 int myNr;
  pthread_mutex_lock( &add_counter_mutex );
 myNr=nr++;
  pthread_mutex_unlock( &add_counter_mutex );
  return myNr;
}

vt_plugin_cntr_metric_info * get_event_info(char * event_name){
 vt_plugin_cntr_metric_info * return_values;
  return_values=
   malloc(2 * sizeof(vt_plugin_cntr_metric_info) );
 return_values[0].name=strdup(event_name);
 return_values[0].unit=NULL;
 return_values[0].cntr_property=VT_PLUGIN_CNTR_LAST|VT_PLUGIN_CNTR_ACC|
     VT_PLUGIN_CNTR_UNSIGNED;
 return_values[1].name=NULL;
 return return_values;
}
uint64_t get_value(int counterIndex){
 return 0ULL;
}
void fini(){
}

vt_plugin_cntr_info get_info(){
 vt_plugin_cntr_info info;
  memset(&info,0,sizeof(vt_plugin_cntr_info));
```

```
 info.init               = init;
 info.add_counter        = add_counter;
  info.vt_plugin_cntr_version = VT_PLUGIN_CNTR_VERSION;
 info.run_per            = VT_PLUGIN_CNTR_PER_THREAD;
 info.synch              = VT_PLUGIN_CNTR_SYNCH;
 info.get_event_info     = get_event_info;
 info.get_current_value = get_value;
 info.finalize           = fini;
 return info;
}
```

Listing D.1: Simplistic synchronous per-thread plugin

```
#include <vampirtrace/vt_plugin_cntr.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <linux/perf_event.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/ioctl.h>
#include <pthread.h>


static int nr=0;

static pthread_mutex_t add_counter_mutex;

uint64_t (*wtime)(void);

int nr_events=0;

double start_time;

void set_pform_wtime_function(uint64_t (*pform_wtime)(void))
{
 wtime=pform_wtime;
}

int32_t init(){
  /* check if pthread mutex can be created */
  start_time=wtime();
  return pthread_mutex_init( &add_counter_mutex, NULL );
}


int32_t add_counter(char * event_name){
 int myNr;
  pthread_mutex_lock( &add_counter_mutex );
 nr_events=atoi(event_name);
 myNr=nr++;
  pthread_mutex_unlock( &add_counter_mutex );
  return myNr;
}

vt_plugin_cntr_metric_info * get_event_info(char * event_name){
 vt_plugin_cntr_metric_info * return_values;
  return_values=
   malloc(2 * sizeof(vt_plugin_cntr_metric_info) );
 return_values[0].name=strdup(event_name);
 return_values[0].unit=NULL;
 return_values[0].cntr_property=VT_PLUGIN_CNTR_LAST|VT_PLUGIN_CNTR_ACC|
     VT_PLUGIN_CNTR_UNSIGNED;
 return_values[1].name=NULL;
```

```
 return return_values;
}

uint64_t get_all_values(int32_t counter, vt_plugin_cntr_timevalue ** result_vector↩
    ){
 int i;
 double end_time=wtime();
 vt_plugin_cntr_timevalue * results=malloc(nr_events*sizeof(↩
     vt_plugin_cntr_timevalue));
 for (i=0;i<nr_events;i++)
 {
  results[i].timestamp=start_time+(double)i*(end_time-start_time)/(double)↩
      nr_events;
  results[i].value=i;
 }
 *result_vector=results;
 return nr_events;
}
void fini(){
}

vt_plugin_cntr_info get_info(){
 vt_plugin_cntr_info info;
  memset(&info,0,sizeof(vt_plugin_cntr_info));
 info.init               = init;
 info.add_counter        = add_counter;
  info.vt_plugin_cntr_version = VT_PLUGIN_CNTR_VERSION;
 info.run_per            = VT_PLUGIN_CNTR_PER_THREAD;
 info.synch              = VT_PLUGIN_CNTR_ASYNCH_POST_MORTEM;
 info.set_pform_wtime_function =set_pform_wtime_function;
 info.get_event_info     = get_event_info;
 info.get_all_values     = get_all_values;
 info.finalize           = fini;
 return info;
}
```

Listing D.2: Simplistic asynchronous post-mortem per-thread plugin

```
#include <inttypes.h>
#include <stdio.h>
#include <string.h>
#include <scorep/SCOREP_SubstratePlugins.h>


/* An enter event has been received from Score-P */
static void
enter_region(
    struct SCOREP_Location* location,
    uint64_t                timestamp,
    SCOREP_RegionHandle     regionHandle,
    uint64_t*               metricValues )
{
}

/* An exit event has been received from Score-P */
static void
exit_region(
    struct SCOREP_Location* location,
    uint64_t                timestamp,
    SCOREP_RegionHandle     regionHandle,
    uint64_t*               metricValues )
{
}
```

```
/* Register event functions */
static uint32_t
get_event_functions(
    SCOREP_Substrates_Mode         mode,
    SCOREP_Substrates_Callback** returned )
{
    SCOREP_Substrates_Callback* functions = calloc( SCOREP_SUBSTRATES_NUM_EVENTS,
                                                    sizeof( ↩
                                                        SCOREP_Substrates_Callback↩
                                                         ) );

    /* Only print region events when recording is enabled */
    if ( mode == SCOREP_SUBSTRATES_RECORDING_ENABLED )
    {
        functions[ SCOREP_EVENT_ENTER_REGION ]       = ( SCOREP_Substrates_Callback↩
            )enter_region;
        functions[ SCOREP_EVENT_EXIT_REGION ]        = ( SCOREP_Substrates_Callback↩
            )exit_region;
    }

    *returned = functions;
    return SCOREP_SUBSTRATES_NUM_EVENTS;
}

/* Define plugins and some plugin functions */
SCOREP_SUBSTRATE_PLUGIN_ENTRY( min )
{
    SCOREP_SubstratePluginInfo info;
    memset( &info, 0, sizeof( SCOREP_SubstratePluginInfo ) );
    info.plugin_version       = SCOREP_SUBSTRATE_PLUGIN_VERSION;
    info.get_event_functions = get_event_functions;
    return info;
}
```

Listing D.3: Simplistic substrate plugin

E Abbreviations

Someone with an obsession for arranging things in alphabetical order was an abcedist, whereas someone with an obsession for arranging them in reverse alphabetical order was a zyxedist., Optimus Yarnspinner
The City of Dreaming Books by Walter Moers

ACPI Advanced Configuration and Power Interface.

AMD Advanced Micro Devices.

APM Application Power Management.

APM Advanced Power Management.

ARM Advanced RISC Machine.

AVX Advanced Vector Extensions.

BIOS Basic Input/Output System.

BLAS Basic Linear Algebra Subprograms.

BMC Board Management Controller.

CAF Coarray Fortran.

CFD computational fluid dynamics.

CG conjugate gradiants.

CKE Clock Enable.

CNF-EFG complete nested functions event flow graph.

COSMO Consortium for Small-scale Modeling.

CPU Central Processing Unit.

CSR Configuration Space Register.

CUDA Compute Unified Device Architecture.

DAQ Data Acquisition.

DC direct current.

DCiE datacenter infrastructure efficiency.

DCT Dynamic Concurrency Thottling.

DFS Dynamic Frequency Scaling.

DRAM Dynamic Random Access Memory.

DVFS Dynamic Voltage and Frequency Scaling.

DVS Dynamic Voltage Scaling.

EDP energy-delay product.

EFG event flow graph.

EPB Energy Performance Bias.

ETS energy to solution.

FE finite element.

FFT Fast Fourier Transform.

FLOPS Floating Point Operations Per Second.

FPGA field programmable gate array.

GCC GNU Compiler Collection.

GPGPU general purpose graphics processing unit.

GPU Graphics Processing Unit.

GTI Generic Tools Infrastructure.

HDEEM High Definition Energy Efficiency Monitoring.

HPC High Performance Computing.

I/O input/output.

IBM International Business Machines.

IEEE Institute of Electrical and Electronics Engineers.

IPMI Intelligent Platform Management Interface.

ISA instruction set architecture.

ISO International Organization for Standardization.

IVR integrated voltage regulator.

MBVR main board voltage regulator.

MCDRAM Multi-Channel DRAM.

MIMD multiple-instruction stream, multiple-data stream.

MISD multiple-instruction stream, single-data stream.

MKL Math Kernel Library.

MPI Message Passing Interface.

MSR Model Specific Register.

NI National Instruments.

NPB NAS Parallel Benchmark.

NUMA non-uniform memory access.

OMIS On-line Monitoring Interface Specification.

OpenMP Open Multi-Processing.

OS operating system.

PAPI Performance API.

PARADISO Parallel Direct Sparse Solver Interface.

PCI Peripheral Component Interconnect.

PCPS Per Core P-states.

PDES parallel discrete-event simulations.

PGAS partitioned global address space.

PLL phase-locked loop.

PMC Performance Monitoring Counter.

PMPI MPI Profiling Interface.

PMU Performance Monitoring Unit.

POSIX Portable Operating System Interface.

PSU Power Supply Unit.

PUE Power Usage Effectiveness.

PVM Parallel Virtual Machine.

RAPL Running Average Power Limit.

RDMA Remote Direct Memory Access.

SHMEM Symmetric Hierarchical Memory.

SIMD single-instruction stream, multiple-data stream.

SISD single-instruction stream, single-data stream.

SLURM Simple Linux Utility for Resource Management.

SPEC Standard Performance Evaluation Corporation.

SPECS SPECtral bin cloud microphysicS.

SPMD single-program, multiple-data.

SRAM Static Random Access Memory.

SSE Streaming SIMD Extensions.

SVI Serial VID Interface.

SVID Serial Voltage Identification.

TAU Tuning and Analysis Utilities.

TDP Thermal Design Power.

TUE Total-Power Usage Effectiveness.

UEFI Unified Extensible Firmware Interface.

UFS Uncore Frequency Scaling.

UPMC Uncore Performance Monitoring Counter.

Vampir Visualization and analysis of MPI resources.

VML Vector Mathematical Functions.

VR voltage regulator.

F Glossary and Nomenclature

What other option did I have, now that words had failed me? What do any of us have when words fail us?, Kvothe
The Wise Man's Fear by Patrick Rothfuss

action A function that is executed when a specific status is encountered. For example, the status could be written to a file, or some tuning could be applied based on the status. More details in Section 5.2 .

back end A back end is called by the integration and consumes the measured status of an element group. Based on this status the back end execute specific actions, e.g., storing the status for profiling or tracing. More details in Section 5.2 .

balancing-based tuning Tuning strategy where paths in a parallel program that are not on the critical path can be slowed down until they become the critical path. More details in Section 2.7.

C-state ACPI Power State. Typically implemented using clock gating or power gating. More details in Section 2.6 .

compute node The set of all devices that share the same physical address space and run the same operating system (OS) instance. More details in Section 2.1.

computing system Multiple compute nodes can be connected via network interfaces to communicate with each other and to be able to solve algorithms in a distributed way. I call a constellation of connected compute nodes a *computing system*. More details in Section 2.1.

core Processor *cores* are independent processing units that read and execute instructions. They hold a number of computational units, internal data storage, caches, and additional devices like hardware prefetchers that are attributed to specific cache levels. More details in Section 2.1.

die A processor can host multiple *dies*, where each die is made of a single piece of semiconductor material. More details in Section 2.1.

element group Observable elements can be grouped. This can be used to describe more complex devices (e.g., modern multi-core processors) or software structures (e.g. multi-threaded processes). Specific element groups are hardware element groups E_h, software element groups E_s which only contain hardware, resp. software elements. Element groups with only element ($|E| = 1$) are denoted E^1. Element groups with multiple elements are denoted as E^+. A group with all possible elements of a monitorable computing system is denoted as $\bar{E}$. Element groups have monitorable properties, whose current values represent the status of the element group. More details in Section 5.1 .

event An event is generated by an event generator and interrupts the workload so that the current status of the hardware and software environment can be captured and processed. More details in Section 5.2 .

event generator An event generator defines a transition rule under which the workload is interrupted and processed by a front end. The event generator can be hardware based (most commonly an interrupt) or software based (most commonly instrumentation). More details in Section 5.2 .

front end A front end uses a set of event generators to interrupt a monitored workload. When one of them is triggered (at each event), the front end collects related status elements as an initial status and passes this status to the integration. More details in Section 5.2 .

hardware thread Core resources are shared by multiple *hardware thread*s if the processor supports hardware multithreading. If it does not, a core supports only one hardware thread. More details in Section 2.1.

integration At each event, the integration receives a status from a specific front ends. The integration calls additional status element collectors and adds the received information to the initial status. The status is then passed to the back ends. More details in Section 5.2 .

Message Passing Interface A programming interface for process parallel applications. More details in Section 2.2.

metric A property that only allows numerical values. More details in Section 5.1 .

module Cores can share resources that are not part of the uncore. If a set of cores does share such resources as well as the connection to the uncore, I call the union of cores and core-shared resources a *module* in accordance with the AMD nomenclature. More details in Section 2.1.

observable element An observable element e represents one indivisible software instance e_s (e.g., a thread) or hardware device e_h (e.g., a core that does not support simultaneous multi-threading or an uncore device). Observable elements can be grouped into element groups. More details in Section 5.1 .

Open Multi-Processing A programming interface for thread parallel applications. More details in Section 2.2.

P-state ACPI Performance State. Typically implemented using Dynamic Voltage and Frequency Scaling (DVFS) or Dynamic Frequency Scaling (DFS). More details in Section 2.6 .

performance-based energy efficiency tuning Changing hardware and software parameters to lower the runtime of code regions and thereby increasing the energy efficiency. More details in Section 2.7.

power-based energy efficiency tuning Changing hardware and software parameters to lower the power consumption of code regions and thereby increasing the energy efficiency. More details in Section 2.7.

processor A non-separable physical entity that hosts at least one core and is attached to a mainboard.

property Element groups can be described by their properties. These have a given value range $V = \{v\}$. This value can also be unknown. For example, one software property could be the content of a specific 64 bit register. Thus, the available value range is $V = \{v \in \mathbb{N}_0 \wedge v < 2^{64} \vee v = ?\}$. Initially (when the thread is started) the value of the register is unknown ($?$), however as soon as it is written the first time, it is set to a natural number smaller than 2^{64}. A given property of a specific element group over a specific time period is a status element. A property with a numerical values is called a metric. More details in Section 5.1 .

region-based tuning Tuning strategy that targets specific code regions, for example synchronization routines. When these functions are executed, a tuning mechanism makes use of ACPI states to lower the power consumption without decreasing performance significantly. More details in Section 2.7.

scheduling status The scheduling status $S_{sched}(E^1_{h_a}, t_s, t_e)$ of a single active hardware element $E^1_{h_a}$ over a specific time frame $t_s - t_e$ describes which single software elements E^1_s are scheduled. More details in Section 5.1 .

statistical status element A statistical description of a status element $\tilde{s}(E, f, t_s, t_e, \pi, v)$, where f is a statistical function, e.g., mean, median, or sum. More details in Section 5.2 .

status The status $S(E, t_s, t_e)$ of an observable element group E in a certain time frame $t_s - t_e$ consists of the status elements of all its sub sets within this time frame. There can be only one status element at each time step that describes a specific property of an element group. More details in Section 5.1 .

status element A status element $s = (E, t_s, t_e, \pi, v)$ applies a a value v to a property π, ranging from a given a start time t_s to an end time t_e. More details in Section 5.1 .

status element collector A status element collector is able to provide a set of status elements at specific points in time. More details in Section 5.2 .

T-state ACPI Throttling State. Typically implemented using clock modulation. More details in Section 2.6 .

time step Every element group has specific time steps $\{t\}$ at which they can be stopped and observed. Different elements can have different time steps (e.g., a processor core's status changes with every cycle, a thread's status changes with each instruction). More details in Section 5.1 .

transition Whenever a property of an element group E_x satisfies the condition of a transition rule, a transition $x = (E, t_n, \pi, v, E_x)$ is initiated. This transition marks the end and the beginning of a status element in the same or another element group E. Transitions in an element group are described as $X(E) = \{x = (E, *, *, *, *)\}$. More details in Section 5.1 .

transition rule A transition rule describes a change of a property within a specific element group. More details in Section 5.1 .

uncore In addition to processor inter-connect, processor dies can contain memory controllers, cores, and other devices. I call the union of inter-connect, memory controller, and other devices *uncore*, in accordance with the Intel nomenclature. The AMD term for the uncore is "northbridge", as it represents devices that have previously been external to the processor. More details in Section 2.1.

workload A workload is an element group that is actively observed by a monitoring infrastructure. While monitoring, a monitoring infrastructure can read the internal status of the element group. More details in Section 5.1 .

Acknowledgments

I want to express my gratitude to all the people that made the creation of this thesis possible.
First and foremost, I would like to thank my doctoral supervisor Prof. Dr. Wolfgang E. Nagel, whose teaching, support, advice, and suggestions laid the cornerstones for this work and my academic progress. I would also like to thank Dr. Stefan Pflüger, who introduced me to the field of performance analysis and supported my early academic years, and Dr. Ralph Müller-Pfefferkorn for finding the right balance between leaving me room for my research and being available when I needed his advice.
My gratitude goes to my advisors and research colleagues. Dr. Andreas Knüpfer significantly influenced the shape and content of this thesis. I also thank Daniel Hackenberg for being challenging and advocating at the same time, Thomas Ilsche and Mario Bielert for advices and proof-reading, and Daniel Molka, whose perfectionism can be demanding but significantly improved the quality of this thesis.
Matthias Jurenz, Ronny Tschüter, and Bert Wesarg provided support for integrating changes to the performance monitoring tools VampirTrace and Score-P.
I also thank the funding agencies for granting projects that supported this thesis: the Bundesministerium für Bildung und Forschung (BMBF) with the projects Cool Silicon and Score-E, the Deutsche Forschungsgesellschaft (DFG) with the Collaborative Research Center HAEC, Bull SAS with the project HDEEM, and the European Commission with the Horizon 2020 project READEX.
Last but not least, I want to thank my family for their support: my mother Karen always had my back and supported me. My deepest gratitude goes to my lovely wife, the only biologist who read all of my theses. Thank you for your support and that you never complained when I stayed in office longer or spent weekend days there while leaving you with the kids. I also thank my children Leandra and Kilian for cheering me up when I needed it.