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. Second, I lay out a model that can be used to define and describe energy efficiency tuning on program region scale. In a third step, I present methods to evaluate common power saving mechanisms and list findings for different x86 processors. Fourth, I describe the structure for an integrated infrastructure for measuring and tuning energy efficiency. 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.

1 Introduction

In the last decades, computer simulations have become an indispensable tool to research the behavior of complex systems [Vet15]. 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. 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 [SHeF06] [FS09]. Figure 2 shows the power consumption of the systems depicted in Figure 1. 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 [HOMST13]. However, most HPC applications do not utilize the hardware to the same extend. Therefore, their power consumption is typically lower [HSM+10].

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
Figure 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) systems as well as the aggregated performance (Sum) of all listed systems.

Figure 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 beginning 11/2007.

variability in the power consumption of nodes\(^1\). The SPECpower_ssj 2008 benchmark \(^2\) measures this variability by issuing different levels of load to the tested system. In Figure 3, I compare the first tested dual-socket server by the vendor Fujitsu\(^3\) with a more recent one. The older system\(^4\) 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 \(^5\). 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.

---

\(^1\)A node is a 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) Results of a 2007 dual socket Fujitsu Siemens system

(b) Results of a 2016 dual socket Fujitsu system

Figure 3: Performance and power consumption of two dual socket Intel x86 systems for the SPECpower_ssj benchmark depending on the system load.
This document is structured as follows: I introduce existing technology in Section 2. This includes an overview on contemporary performance and power measurement infrastructures and tuning tools. In Section 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. Some of the hardware parameters are detailed in Section 4 where I list the costs for applying different power saving mechanisms. 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 Section 5. Furthermore, I describe the implementation of several interfaces that enable energy efficiency measurement and tuning. This includes the linking of power and performance measurement and the capability of tools to act on software events. I evaluate the tuning implementations on an x86-based HPC system in Section 6 and show that the proposed solution can indeed be used to tune parallel applications. Section 7 concludes this thesis, summarizes the findings, and gives an outlook on future developments in hard- and software.

2 Background and Related Work

Performance and energy measurement and optimization has been a target of scientific work for decades. In this section, I examine aspects of measuring and optimizing performance and energy efficiency. To do so, I split this field into the measurement of performance and power (in Sections 2.1 and 2.2) and the tuning of energy efficiency (Section 2.3).

2.1 Performance Monitoring

In [Jai91, Chapter 3], Jain describes three different approaches to performance analysis: analytical modeling, simulation, and measurement. In my thesis, I focus on the third method, measurement, which has two major shortcomings, 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 my thesis, I focus on the measurement and tuning of the software/hardware environment of existing HPC systems. Here, 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 distinguishes two measurement techniques: “tracing” and “sampling” [Jai91, Section 8.1.1]. 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”. In [ISSH15], Ilsche et al. describe performance analysis as a three staged process, based on the PhD thesis of Juckeland [Juc12, Chapter 3]. Their nomenclature, which I use in this document, is depicted in Figure 4. The first layer, data acquisition, represents Jain’s measurement techniques. In the data recording stage, data produced by the data acquisition layer is stored and processed. The final stage, data presentation, targets the representation of the collected data.

In Table 1, I shortly summarize some of the numerous performance monitoring tools targeted at parallel applications. More information on such tools can be found in [ISSH15], which I co-authored.

![Table 1: Performance monitoring tools](image)

<table>
<thead>
<tr>
<th>Performance monitoring tool</th>
<th>Data Acquisition</th>
<th>Data Recording</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score-P [KRaM+12]</td>
<td>Instrumentation, Sampling, Profiles, Timelines</td>
<td></td>
</tr>
<tr>
<td>VampirTrace [MKJ+07]</td>
<td>Instrumentation, Sampling, Profiles, Timelines</td>
<td></td>
</tr>
<tr>
<td>Scalasca 1.x [GWW+10]</td>
<td>Instrumentation, Profiles</td>
<td></td>
</tr>
<tr>
<td>Extrae [Bar15]</td>
<td>Instrumentation, Profiles</td>
<td></td>
</tr>
<tr>
<td>HPCToolkit [ABF+10]</td>
<td>Sampling, Profiles, Timelines</td>
<td></td>
</tr>
<tr>
<td>OpenSpeedshop [SGM+08]</td>
<td>Instrumentation, Sampling, Profiles, Timelines</td>
<td></td>
</tr>
<tr>
<td>TAU [SML06]</td>
<td>Instrumentation, Profiles, Timelines</td>
<td></td>
</tr>
<tr>
<td>Alinea MAP [JBOO15]</td>
<td>Sampling, Profiles</td>
<td></td>
</tr>
<tr>
<td>CrayPat [KHO05]</td>
<td>Instrumentation, Profiles, Timelines</td>
<td></td>
</tr>
<tr>
<td>VTune [Rob05]</td>
<td>Sampling, Profiles, Timelines</td>
<td></td>
</tr>
</tbody>
</table>

* Prototype

![Figure 4: Classification of performance analysis techniques by Ilsche et al.](image)
2.2 Power and Energy Modeling and Measurement

In addition to performance monitoring, 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 [HG+15, Section 1]: “temporal and spatial resolution, accuracy, scalability, cost, and convenience”. We also describe that these criteria partially contradict each other. In this section, I list the most prominent power measurement infrastructures used in HPC.

Power consumption information can be retrieved from modeling and measurement infrastructures. The idea of modeling power consumption has been seized by various researchers [MMO01, CM05, IM05, SBM09, GMG+10]. Modeled power information is also implemented in hardware by processor vendors [HIS+13, JSK+10, FAWR+11]. While all of these solutions come with a low cost, a high scalability, a high temporal and spatial resolution, and a potentially high convenience, their accuracy is limited. Thus, if the focus is on accuracy, actual measurements are the better choice.

Several researchers implement low-level instrumentation to analyze the power consumption of individual components [HG+15, GFS+10, LPD13, BLFP10, DHK+15]. At a higher cost, their infrastructures are able to capture accurate power information at a high temporal and spatial resolution, and accuracy. Scalability and convenience differ between the implementations. Alternatively, researchers often use calibrated power meters like Watts up? Pro [Wat] or ZES ZIMMER LMG [ZEST1], whose temporal resolution is limited. Power consumption is also of interest for data center monitoring. Vendors like Bull [HIS+14], Cray [ECCS12], and IBM [KFH+14] implement proprietary solutions with high spatial and temporal resolution. To increase the convenience for accessing power information, several researchers implemented infrastructures that provide a common interface for different power measurement sources, e.g., dataheap [KH12], PMLib [LDB+15], PAPI [WJK+12], and PowerAPI [BNS13].

2.3 Power-based Energy Efficiency Runtime Tuning

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, some of which are described in the ACPI standard [acp16]. 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. Table 3 provides an overview on the classification of the described mechanisms.

Linux offers power saving mechanisms that utilize ACPI states [PS06, PLB07]. However, their heuristics are not optimal, as stated in [SKK11, SH11]. On a coarser scale, whole compute nodes can be shut down when they are not necessary to process the current load [Sch, GJW11]. In HPC, the usage of standard parallelization paradigms like OpenMP and MPI and the Single Program Multiple Data (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, e.g., [MCS+13] and [BPP15], targets specific code regions. When these functions are executed, ACPI states are utilized to lower the power consumption without decreasing performance significantly. 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. Examples are described in [RLJS+09, TLF+12, WSM15].

Table 3: Overview on different power based energy efficiency runtime tuning tools

<table>
<thead>
<tr>
<th>Tuning Tool</th>
<th>Data Acquisition Type</th>
<th>Data Acquisition Scope</th>
<th>Analysis Scope</th>
<th>ACPI Mechanism</th>
<th>Online/Offline</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpuside menu gov.</td>
<td>Sampling</td>
<td>hardware thread</td>
<td>hardware thread</td>
<td>C-state</td>
<td>online</td>
</tr>
<tr>
<td>Green Governors</td>
<td>Sampling</td>
<td>hardware thread</td>
<td>hardware thread</td>
<td>P-state</td>
<td>online</td>
</tr>
<tr>
<td>ps-governor</td>
<td>Sampling</td>
<td>hardware thread</td>
<td>hardware thread</td>
<td>P-state</td>
<td>online</td>
</tr>
<tr>
<td>SKURM (Sch)</td>
<td>Sampling</td>
<td>hardware thread</td>
<td>hardware thread</td>
<td>S-state</td>
<td>online</td>
</tr>
<tr>
<td>Marlowe (GJW11)</td>
<td>Sampling</td>
<td>hardware thread</td>
<td>hardware thread</td>
<td>S-state</td>
<td>online</td>
</tr>
<tr>
<td>Adagio (RLJS+09)</td>
<td>Sampling</td>
<td>hardware thread</td>
<td>hardware thread</td>
<td>S-state</td>
<td>online</td>
</tr>
<tr>
<td>Periscope Tuning</td>
<td>Sampling</td>
<td>hardware thread</td>
<td>hardware thread</td>
<td>S-state</td>
<td>online</td>
</tr>
<tr>
<td>Framework</td>
<td>Sampling</td>
<td>hardware thread</td>
<td>hardware thread</td>
<td>S-state</td>
<td>online</td>
</tr>
<tr>
<td>Renci/UNC</td>
<td>Sampling</td>
<td>hardware thread</td>
<td>hardware thread</td>
<td>S-state</td>
<td>online</td>
</tr>
<tr>
<td>Green Queue</td>
<td>Sampling</td>
<td>hardware thread</td>
<td>hardware thread</td>
<td>S-state</td>
<td>online</td>
</tr>
<tr>
<td>ACTOR [CMAMBN08]</td>
<td>Sampling</td>
<td>hardware thread</td>
<td>hardware thread</td>
<td>S-state</td>
<td>online</td>
</tr>
<tr>
<td>ENAW [WSM15]</td>
<td>Sampling</td>
<td>hardware thread</td>
<td>hardware thread</td>
<td>S-state</td>
<td>online</td>
</tr>
</tbody>
</table>
2.4 Summary and Open Topics

In this section, I 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 Section 3 I describe how energy-efficient configurations for HPC applications can be found. This relies partially on hardware parameters, which are described in Section 4. In addition to that, the application has to be analyzed, which includes a survey of its performance and energy efficiency saving potential. This requires a monitoring tool that is able to capture performance and power information. Such an infrastructure is the topic of Section 5. Furthermore, I extend the infrastructure with the capability to act on software events. This can be used for energy efficiency tuning. In Section 6 I use the infrastructure to tune the energy efficiency of HPC applications.

3 A Model for Energy Efficiency Tuning

Each power saving mechanism has a certain influence on the power consumption of the tuned component and the runtime of the executed software. In the following, I describe such a model that can be used to quantify the effect of different power saving mechanisms on energy efficiency. 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 $c_0$. 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 ($c \leftarrow c_0$), or it is inherited from the previous region ($c \leftarrow c_{\text{prev}}$). Depending on the weighting function $\lambda(r, c) = f_\lambda(P(r, c), t(r, c))$ (e.g., $ETS(r, c) = P(r, c) \times t(r, c)$), 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 1). The throughput substitution can be used for common metrics like energy to solution (ETS) and energy-delay product (EDP) [HIC94] and is valuable when multiple regions with the same behavior are executed.

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

(1)

In the simplest case, depicted in Figure 5(a) only one 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

![Diagram](image1.png)

(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.

![Diagram](image2.png)

(b) The configuration is set to $c_1$ 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.

Figure 5: Comparison between an unoptimized and optimized execution of $r_i$. 

5
\( c_t \) that is reset afterwards, runtime, throughput, and power consumption of \( r_i \) and its successor \( r_{i+1} \) are significantly influenced. This is depicted in Figure 5b. 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 \( t(\text{switch}, c) \) and power consumption \( P(\text{switch}, c) \) 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 ACPI P- and T-states into account. After a specific delay \( d(c_{<}, c) \), 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 \( t(\text{switch}, c_t) \). Runtime and power consumption of the successive region \( r_{i+1} \) are also influenced from the optimized configuration \( c_t \). In Figure 5b, the throughput of \( r_{i+1} \) is decreased while the hardware is still in \( c_t \). Thus, the runtime increases by a specific amount \( t_{\text{echo}} \). This prolongation can be calculated by the throughput and runtime of \( r_{i+1} \) under \( c_0 \) and \( c_t \) and \( d(c_t, c_0) \). Equation 2 describes how \( t_{\text{echo}}(r, c_{<}, c) \) performs under different throughput levels for the previous configuration \( c_{<} \) and the target \( c \).

\[
t_{\text{echo}}(r, c_{<}, c) = \frac{p(r, c) - p(r, c_{<})}{p(r, c)} \ast d(c_{<}, c)
\]

The access to change the configuration increases runtime and energy consumption additionally. The total runtime of a region \( r \), when the configuration is switched from \( c_{<} \) to \( c \) is described in (3). Energy values can be calculated accordingly.

\[
t(r, c_{<}, c) = t(\text{switch}, c_{<}) + t(r, c) + \frac{p(r, c) - p(r, c_{<})}{p(r, c)} \ast d(c_{<}, c)
\]

The model that I described in this section has system dependent parameters \( d, t(\text{switch}, c) \), and \( P(\text{switch}, c) \) and software dependent parameters \( t(r, c) \) and \( P(r, c) \). While the former are discussed in Section 4, I discuss the inclusion of software parameters and the application of configurations in Section 5.

4 System Parameters for Energy Efficiency Models

In this section, I shortly describe methods to measure the overhead and the delay of accessing ACPI P- and C-states. I present implementation details of ACPI T-states on Intel processors in [SIB,16].

Achievable power savings significantly depend (1) on the power consumption baseline of a computing node and (2) on the executed software, which defines the processor’s activity factor [Wes,11, Section 5.2.1.2]. Therefore, achievable power savings are not mentioned in this section.

4.1 ACPI P-States

ACPI P-States are typically implemented with dynamic voltage and frequency scaling (DVFS). This is a technique that enables changing the processor’s frequency and voltage at runtime [CSB,92]. The switching time \( t(\text{switch}, c) \) for changing a processor’s P-state significantly depends on the mechanism that is used to access it. The most common library \texttt{libcpufreq} introduces a significant overhead for each frequency transition. Thus, a low level interface should be used to limit \( t(\text{switch}, c) \). The transition delay \( d \) depends on three parameters: the source frequency, the target frequency, and the processor architecture. In [XIL,14], Mazouz et al. described how \( d \) can be measured. On most architectures, it is (1) in the order of tens of microseconds, (2) low for frequency reductions, and (3) increasing with the frequency difference when the frequency is increased. This can be seen in Figure 5a, which depicts the transition latency on a Sandy Bridge based Intel Xeon E5-2670. However, contemporary Intel processors deviate from that behavior, as shown Figure 5b. FTaLaT is not able to capture the responsible effects. In [HSI,15], which I co-authored, we describe which changes have to be made to measure them. On contemporary processors, a transition window is introduced, whose functionality is depicted in Figure 4. Due to this mechanism, \( d \) can vary between 21 and 524 µs, depending on when the frequency change request has been issued.

4.2 ACPI C-States

ACPI C-states are usually implemented via clock gating and power gating. They are activated with specialized instructions like \texttt{halt} or \texttt{mwait}. However, these instructions are typically only available
in an operating system context and cannot be used from user space. Additionally, the processor halts after a processor idle 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. Additionally, \( t(switch, c) \) and \( d \) cannot be distinguished.

In [SMW14], I show how the transition latency \( d \) of a processor core that returns to a non-idling state can be measured. Here, one core wakes up another core, and the respective timings for triggering the wake-up and the return to an active state are captured. Figure 8 visualizes the difference between these times for three different Intel server processors. It is obvious that \( d \) depends on the idle state and the applied frequency.

5 An Infrastructure for Monitoring & Tuning Energy Efficiency

Performance monitoring and energy efficiency tuning tools share a common concept. The workload is regularly interrupted to either monitor the current status or to enhance the efficiency of the system. This can be done either with instrumentation or sampling. When the workload is interrupted, additional information is captured. This could be, for example, the content of performance monitoring counter registers, or the load factor of a processor core. The collected information is then used to either record the data or to tune the system.

Figure 9 depicts an overview on such a unified tool infrastructure. Here, the workload is interrupted by different front ends. These collect front end specific information (e.g., the message size in an MPI instrumentation front end) and pass it to the integration layer. The integration layer is able to collect additional information about the current status of the workload via so called status element collectors. These are specialized parts that are able to collect common information about the monitored software or hardware, which is independent of the front ends. The overall status, which now contains status elements from the front ends and the status element collectors, is then forwarded to the back ends. These either collect the information for recording purposes or use it to classify the system state and change the hardware or software environment in order to tune it.
To make such a tool applicable for a large range of HPC applications, its front ends must support the most common data acquisition methods that target HPC applications. This includes instrumentation front ends for parallelization paradigms and compilers that are used in HPC. Furthermore, it must be able to capture necessary power consumption information and other data that is needed to analyze the energy efficiency of applications. Such a tool must support different back ends targeted to record and tune the energy efficiency of programs. HPC performance monitoring tools as listed in Table 1 already implement all necessary front ends. Furthermore, they are scalable and can thus be used to analyze the behavior of large scale applications. Therefore, I extend existing performance monitoring tools to implement the two missing necessities: the measurement of power information and the possibility to use the collected information to tune the hardware and software environment.

I discuss the inclusion of energy efficiency related information in performance monitoring tools in [STH11]. Here, I implemented an extension for VampirTrace that is able to handle most different metrics. The individual plugins are provided to VampirTrace as shared objects that can be loaded by the dynamic linker at runtime, based on which metrics should be recorded. The extension is called “VampirTrace plugin counters” and provides an interface to implement new plugins in the programming language C. Later, the interface has been ported to Score-P. Changes and peculiarities of this interface are described in [STI17]. In the last years, researchers have implemented numerous plugins to include the most common power measurement infrastructures [STH11, Min13, HIS13, HRD14, HIS14, KFH14, CDKL14].

I introduce the idea of a unified infrastructure for measuring and tuning the energy efficiency of HPC application in [SM13]. In this publication, I show how a common instrumentation could be used to tune the energy efficiency of OpenMP parallel applications. In [STI17], I introduce an extension for Score-P that externalizes run-time events. I also show that it is possible to use this extension for energy efficiency tuning.

Figure 10 depicts the two plugin interfaces in Score-P and introduces the terminology of the Score-P project. Here, front ends, back ends, and status element collectors are called adapters, substrates, and services, respectively.

6 Energy Efficiency Tuning Evaluation

In this section, I shortly present results of two different energy efficiency tuning substrates. Both tests were executed on the haswell partition of the taurus HPC system located at the Center for Information Services and High Performance Computing (ZIH) at the Technische Universität Dresden. Details about the system can be found online.

The individual compute nodes underly specific power and performance variances [SHS16]. To limit the influence of this factor, each benchmark is tested with different configurations within one SLURM job. Thus, all iterations use the same compute nodes. I also repeat every measurement three times and use the result with the median runtime in the discussion. More examples and detailed results are given in the full thesis.

https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/SystemTaurus
6.1 Region-based Offline Tuning

The first substrate uses libadapt \cite{SM13}, which can be used to change hardware and software properties at runtime and has been presented in \cite[Section 4.4]{STI+17}. To gather an energy-efficient configuration, I use a test run where different configurations are applied and the performance and power characteristics are recorded in a trace. This trace is parsed to gather optimal core and uncore frequencies. I test this plugin with the OpenMP parallel Block Tri-diagonal (BT) benchmark, which is part of the NAS Parallel Benchmarks \cite{BBB+94}. The main phase of the benchmark executes a loop of five OpenMP parallel regions: rhs, x_solve, y_solve, z_solve, and add.

I use the model from Section 3 to determine energy-efficient configurations. Based on the projected energy and runtime, more suitable settings are applied to the workload. Here, the core frequency is reduced to 1.4 GHz in regions add and rhs. Likewise the uncore frequency in x_solve is reduced to 1.4 GHz and to 2.8 GHz for others. Figure 11 depicts the power consumption profile of a single loop iteration for the default and the tuned configuration. The energy consumption decreases by 7.6% even though the runtime increases by 2.4%, as shown in Table 5.

![Figure 11: Comparison of the BT power consumption for default (top) and tuned (bottom) configuration.](image)

6.2 Balancing-based Online Tuning

The interfaces that I introduced in Section 5 can also be used for other energy efficiency tuning strategies. I implemented a substrate plugin which mimics the behavior of the Adagio framework, which has been presented by Rountree et al. \cite{RLdS+09}. It determines pairs of compute and synchronization regions. Based on the relative duration of the synchronization phase, it locally detects imbalances. If such an imbalance is detected, the pair of compute and synchronization region is executed at a lower frequency in future iterations until it becomes the critical path. More details can be found in \cite[Section 4.5]{STI+17}.

I demonstrate the applicability of this plugin with GemsFDTD, which stands for General ElectroMagnetic Solvers (GEMS) that are using finite-difference time-domain (FDTD) methods. It is part of the SPEC MPI2007 benchmark suite \cite[Section 3]{MvWL+09}, 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. In Table 6, I show the effects of tracing and balancing on the runtime and energy consumption when running the medium dataset on 120 MPI ranks. Here, the energy consumption is reduced by 14.3%, despite a runtime increase of 4.1%. Figure 12 depicts the MPI latencies and power consumption of the MPI ranks and compute nodes.

![Figure 12: Vampir visualization of MPI latencies and power consumption metrics for an instrumented run and a tuned run of SPEC MPI2007 benchmark 113.GemsFDTD on 120 MPI ranks. Maximal, average, and minimal values of the individual metrics are marked in red, black, and blue, respectively.](image)

### Table 5: Runtime and energy consumption of BT benchmark (OpenMP, Class C, 24 threads)

<table>
<thead>
<tr>
<th>Instrumentation</th>
<th>Runtime [s]</th>
<th>Energy [kJ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>uninstrumented</td>
<td>45.68</td>
<td>13.72</td>
</tr>
<tr>
<td>instrumented, default</td>
<td>45.57</td>
<td>13.96</td>
</tr>
<tr>
<td>instrumented, tuned</td>
<td>46.79</td>
<td>12.85</td>
</tr>
</tbody>
</table>

### Table 6: Runtime and energy costs for Score-P runs of 113.GemsFDTD (120 MPI ranks)

<table>
<thead>
<tr>
<th>Instrumentation</th>
<th>Runtime [s]</th>
<th>Energy [kJ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>no instrumentation</td>
<td>193</td>
<td>271.7</td>
</tr>
<tr>
<td>Score-P Tracing</td>
<td>191</td>
<td>270.7</td>
</tr>
<tr>
<td>Score-P Balancing</td>
<td>201</td>
<td>232.8</td>
</tr>
</tbody>
</table>
7 Summary, Outlook, and Future Work

Summary
In Section 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. In Section 4 I described the hardware parameters for common power-saving states and show how they can be measured. This includes an analysis of the overhead to initiate a new state and the delay until the new state is applied. 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 and discussed its application to existing monitoring tools in Section 5. To do so, I implemented an interface that can be used to augment traditional performance information with power and energy-related data. This interface 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. A second interface enables users to tune the energy efficiency of programs, alternatively to just measuring it. In the final section, I evaluated two tuning approaches by applying them to HPC benchmarks and applications. The first approach implements a region-based offline tuning that uses the model from Section 3 to determine energy-efficient configurations for program regions. 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.

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”. 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 tries 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 dim silicon spans a complex decision tree for processor architects as well as software architects. 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 to further configure the available mechanisms. 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. It can be used to include most different performance metrics, which supports the evaluation and helps to pinpoint bottlenecks. Furthermore, it 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 can pose a problem. To tackle such limitations, Patki et al. suggest that one “should also consider overprovisioning for high-performance computing (HPC)” [PLR+13]. Here, “full power is guaranteed to a restricted number of nodes (worst case provisioning)” or “power is limited to more nodes (overprovisioning)”. 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.

Future Work
Apart from energy efficiency, the established interfaces also enable new options for performance analysis and debugging. One example has been the provision of event flow graphs as described in STI+17. 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. A future development of the metric interface could enable more fine-grained hardware scopes. 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 both interfaces makes it easy to implement such extensions and enable new research in the areas of performance and energy efficiency evaluation and tuning.
References


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


[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


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


[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

