Lehrstuhl für Rechnerarchitektur & Parallele Systeme
Fakultät für Informatik
Technische Universität München

Automatic Analysis and Exploitation of Dynamism for Energy-Efficient HPC Applications

Madhura Kumaraswamy

Lehrstuhl für Rechnerarchitektur & Parallele Systeme
Fakultät für Informatik
Technische Universität München

Automatic Analysis and Exploitation of Dynamism for Energy-Efficient HPC Applications

Madhura Kumaraswamy

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 03.08.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 13.09.2020 angenommen.

Abstract

The increase in the energy consumption as a result of growing computational performance is a major challenge to reach Exascale levels of computing due to the cooling costs of High Performance Computing (HPC) systems. Unfortunately, developers focus on improving the performance of existing algorithms, and neglect potential improvements to energy-efficiency due to the lack of platform knowledge. To overcome these challenges, we propose a dynamic autotuning approach constituting design-time and runtime stages by combining methods from the embedded systems and HPC domains to optimize the energy consumption. The approach targets applications that exhibit changing characteristics, known as *application dynamism* between individual iterations of the time loop as well as within a single time loop.

We leverage the inter-loop and intra-loop dynamism using two tuning plugins, which use a search strategy to create a search space of different configurations of tuning knobs. To analyze the inter-loop dynamism, we propose *features* to cluster loops with similar characteristics using the DBSCAN and spectral clustering methods. To prevent the search space from exploding and to save tuning time and cost, we optimize the search space using a probability distribution based on a Gaussian kernel to test a large number of configurations in a single application run. Our proposed approach stores the best configuration for each instance of a code region based on its unique computational characteristics in a tuning model.

In the runtime tuning stage, we use the tuning model to dynamically switch the system configuration for different code regions. To predict the behaviour of unseen loops during production runs, we implemented predictors based on a second-order Markov chain, one-bit and two-bit dynamic branch prediction schemes. While the proposed method supports an automatic approach, we also enable the specification of domain knowledge to improve the tuning results. We evaluated our methodology using three real-world applications and one proxy application, and achieved up to 20% improvement in both energy-efficiency and performance.

Acknowledgments

Years of research have culminated in this dissertation, which would not have been possible without the support and guidance of the following people.

First of all, I would like to express my deepest gratitude to my advisor Prof. Dr. Michael Gerndt for giving me this wonderful and fruitful opportunity to pursue research in a stimulating and dynamic research environment. I am thankful for his immense patience, wisdom, support, insightful guidance, and constructive criticism that equipped me with the confidence to independently develop research ideas.

I am also grateful to my second advisor Prof. Per Gunnar Kjeldsberg for his valuable and expert feedback during my visit to the Norwegian University of Science and Technology (NTNU).

The work presented in this dissertation is an extension of the core concepts developed in the READEX project. I would like to thank all the researchers who worked on the READEX project and directly or indirectly helped me throughout my journey. I am also indebted to the Centre for Information Services and High Performance Computing (ZIH) at the Technische Universität Dresden for providing me the resources and enabling me to pursue my work. I gratefully acknowledge the HPC Support team for their patience with my constant stream of emails.

I would also like to thank Prof. Dr. Martin Schulz, the head of the Chair of Computer Architecture and Parallel Systems (CAPS) at the Technische Universität München (TUM) for creating a great working environment, and encouraging collaboration between other PhD students. I am grateful to Andreas Wilhelm, Anamika Chowdhury, Robert Mijaković, Vladimir Podolskiy, and Dai Yang for the fruitful discussions and fun moments at the Chair. I also thank TUM for the financial support for my research during the last years of my PhD.

I express my appreciation and thanks to all my friends for their encouragement that kept me going through tough times. Last but not least, I am incredibly grateful to my family for their moral support and love, and for inspiring me to pursue my dreams. Special thanks go to my wonderful sister and friend who gave me the much required moments of comic relief. Thank you all.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

High Performance Computing (HPC) systems are designed to achieve peak performance to enable scientific and industrial advancements. Even as the advancements in semiconductor technology have increased the number of transistors on a die, Moore's law has hit the power wall [1]. With growing computational performance, there is typically also an increase in a system's energy consumption, which in turn is a major driver for the Total Cost of Ownership (TCO) of HPC systems. With Dennard scaling coming to an end, power density is now becoming a problem as more transistors are crammed together on a single die, thus reducing the effectiveness of cooling capabilities, resulting in power-bound exascale HPC systems. This is a cause for concern because HPC centers now face the challenge of operating the hardware with the power or energy consumption being a prominent contributor to the TCO [2]. Hence, it is likely that users will be forced to optimize their software with respect to energy-efficiency.

Table 1.1 shows the top four systems in the June 2020 Top500 list. The top ranked system Fugaku, which is located at the RIKEN Center for Computational Science, Japan, has a maximum performance of 415.5 PFlop/s and consumes 28.3 MW of power, which is more than the power budget set by the US Department of Energy for Exascale machines. The future of supercomputing is in developing Exascale systems. To scale the current systems to reach the Exascale level, we would require more than 50 MW of power. Thus, the challenges related to power and energy consumption are a roadblock in achieving this milestone. Therefore, advances in all areas ranging from the hardware technology to the operating mechanisms such as cooling are required to optimize the overall energy consumption of current and future HPC systems.

1.1 Motivation and Problem Specification

HPC applications typically exhibit changing characteristics, known as *application dynamism* in different regions of the program. The program execution may jump between compute, memory, and I/O-bound code regions due to changes in the workload at runtime. Examples of dynamism can be found in many current HPC applications, including weather forecasting, molecular dynamics and adaptive mesh-refinement applications. Figure 1.1 illustrates the trend of the execution time across the individual time steps of the main progress loop, known as *phases* of INDEED [3], a sheet-metal forming simulation application. INDEED is a highly dynamic application that performs an adaptive mesh refinement when different tools come in

Table 1.1: Power consumption and *RMax* (maximum LINPACK performance achieved) for the top four supercomputers in the Top500 list as of June 2020.

Rank	System	Location	Cores	RMax (PFlop/s)	Power (MW)
1	Fugaku	RIKEN Center for Computational Science, Japan	7,299,072	415.5	28.3
2	Summit	DOE/SC/Oak Ridge National Laboratory, United States	2,414,592	148.6	10.1
3	Sierra	DOE/NNSA/LLNL, United States	1,572,480	94.6	7.44
4	Sunway TaihuLight	National Supercomputing Center/Wuxi, China	10,649,600	93.0	15.4

contact with the workpiece. The first spike in the graph occurs when the tool makes initial contact with the workpiece, and subsequent spikes are produced when the tool comes in contact with more refined regions. The simulation adapts the time step to keep the problem tractable while refining the mesh.

Such varying characteristics between individual simulation loops of HPC applications offer the potential to perform tuning to improve the energy-efficiency by leveraging their dynamic behaviour. Individual phases can be grouped by characterizing the behaviour using metrics, or *features*, and different optimal configurations can be selected for each group of phases. However, detecting this dynamism manually is a time-consuming and painstaking task that demands severe programming effort and domain-level expertise.

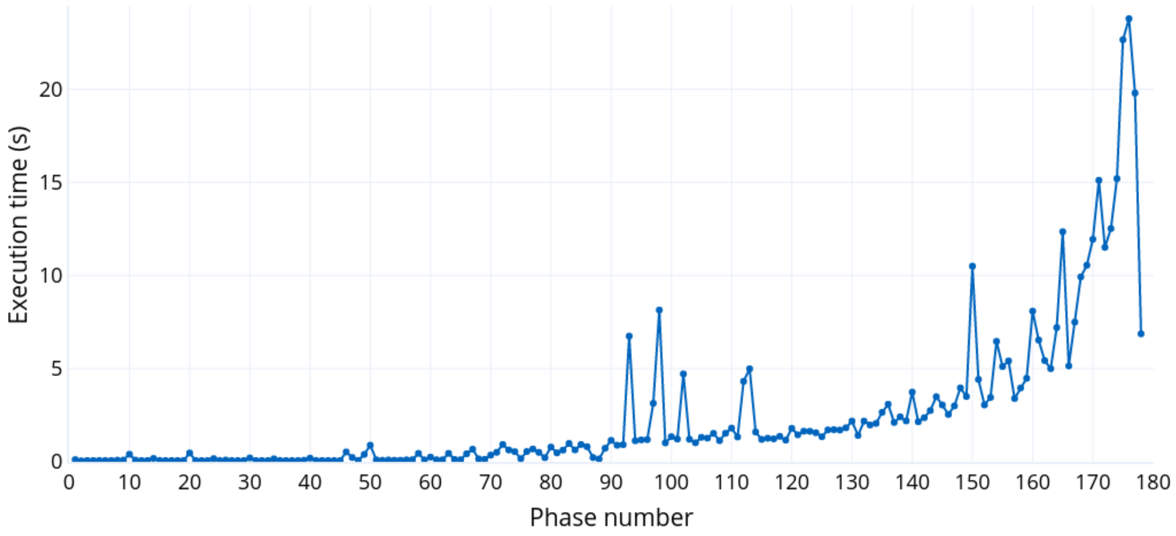


Figure 1.1: Variation of the execution time across the phases of INDEED. The variation arises from an adaptive mesh refinement performed when the number of contact points between the tools and the workpiece increases.

Another challenge is that developers focus on improving algorithms with regards to accuracy and performance while neglecting possible improvements to energy-efficiency [4] due to the lack of platform and hardware knowledge required to exploit application dynamism. Several power optimization techniques such as clock gating, clock modulation and power gating have already been implemented in today's processors by hardware vendors. Modern processor architectures also provide the possibility to adjust the clock frequency of the CPU at runtime using Dynamic Voltage Frequency Scaling (DVFS) to improve the energy-efficiency [5]. During the execution of HPC codes, the cores are not always busy, for example, when the program execution moves from a compute-bound to a memory-bound code region. During this time, the presence of memory requests between the last-level cache and the RAM indicates that the frequency can be safely throttled down and the voltage reduced without affecting the execution time.

For earlier Intel processor architectures such as Nehalem-EP and Westmere-EP, the frequency of the uncore components such as the L3 cache and the interconnect was fixed. For the Sandy Bridge-EP and Ivy Bridge-EP architectures, a common frequency was set for the core and uncore parts. Intel has introduced a new feature called Uncore Frequency Scaling (UFS) starting from the Haswell processors, which allows the setting of separate frequencies for the core and the uncore domains independently [6]. Previous research [5] has also shown that changing the uncore frequency can significantly impact the energy consumption.

Manually setting these tuning knobs, or the so-called *tuning parameters* for different program regions is difficult as it requires application domain knowledge. One solution is to use an automatic optimization approach called autotuning, which automatically generates a search space of possible combinations of tuning parameters, and evaluates them using experiments to identify the best *system configuration* [7], which is the set of the best settings of the tuning parameters.

Most existing tools leverage the hardware features by relying on static tuning [8], where a single frequency is set for the entire application run based on the current workload, with the goal of finding a single optimal configuration. Such a coarse-grained approach has the drawback that it produces a suboptimal solution since it fails the take advantage of the tuning potential of individual program regions. Hence, our goal is to implement a more fine-grained approach, which selects the best configuration for each instance of a code region based on its unique computational characteristics.

1.2 Solution: A Tools-Aided Approach to Optimize the Energy-Efficiency

The process of determining the best system configuration can be done by using a brute-force strategy, known as an exhaustive search. The exhaustive search strategy walks the full search space and explores all the combinations of the tuning knobs. While this may result in the optimal system configuration, it is not practical to run HPC applications for hours or even days to find the perfect solution. Thus, a key aspect in autotuning is the trade-off between quality of the results and the effort spent in searching for the best solution. An alternative method is to use a random search strategy, which selects valid points in the search space randomly. Both the random and exhaustive search strategies are good for autotuning when the structure of the objective function [9] and the application behavior in response to the changing configurations are unknown.

To automate the tuning process and free the user from learning the intrinsics of autotuning, we present a tools-aided approach by combining several pre-existing tools with novel runtime tuning techniques to guide the optimal tuning of the HPC stack constituting the hardware, the runtime system, and the application-level tuning parameters. The work done in this thesis is an extension of the READEX (Runtime Exploitation of Application Dynamism for Energy-efficient eXascale computing) [4] methodology, which aimed to automate the process of determining the best system configurations for different program regions by developing a tools-aided methodology for dynamic autotuning. READEX combines technologies from two ends of the computing spectrum: system scenario methodology from the embedded world and autotuning from the HPC domain.

READEX is a two-staged approach consisting of Design-Time Analysis (DTA), and Runtime Application Tuning (RAT), with Score-P [10] as the common instrumentation and measurement infrastructure. At design-time, different instances of the program regions, known as *runtime situations* (rts's) are detected during the iterations of the main progress loop, or phases. Optimized system configurations are then determined for the phases as well as the rts's, based on the type of dynamism that is exhibited by the application. The variation in the application behaviour and characteristics may arise between individual phases or within a single phase. Rts's with the best system configurations are grouped into *scenarios*, and encapsulated in a *tuning model*. At runtime, the tuning model is used to guide the dynamic switching of system configurations. To improve the automatic tuning results, application experts may also specify domain knowledge in the form of annotations to define the application structure, characteristics and the Application Tuning Parameters (ATPs).

The tuning methodology begins with a series of pre-analysis steps to determine if tuning the application potentially results in energy savings. First, the *scorep-autofilter* tool filters out all fine-granular regions that generate measurement overhead. Then, the *readex-dyn-detect* tool determines *significant regions*, which are a subset of all instrumented regions that are coarse-granular enough for dynamic switching of the tuning parameters. It also determines if the characteristics of the application vary between different phases due to the change in the control flow, or within a single phase due to the changing characteristics between individual rts's that are called in that phase. The findings of *readex-dyn-detect* are stored in a configuration file that is used during DTA.

Design-Time Analysis (DTA) is performed by the Periscope Tuning Framework (PTF), which was developed at the Technische Universität München, Germany, and is a distributed framework based on an extensible and modular architecture implemented using a hierarchy of analysis agents. PTF invokes tuning plugins to tune different aspects of HPC applications. The *intraphase* tuning plugin is used to exploit intra-phase dynamism, while the novel *interphase* plugin is called when inter-phase dynamism is detected for an application. Both plugins find the best settings of the hardware tuning parameters (CPU and uncore frequency), and system software tuning parameters (number of OpenMP threads) for different objectives such as energy consumption, execution time, Energy Delay Product (EDP), Energy Delay Product Squared (ED2P), TCO, and also their normalized versions. The *intraphase* tuning plugin additionally supports the tuning of ATPs. It first determines their optimal settings, and then tunes the hardware and system software tuning parameters using a user-defined search strategy. Finally, it determines a single best system configuration for all the phases of the application, and rts-best configurations for individual rts's of the significant regions.

While the *intraphase* plugin determines optimal configurations for the phase and the rts's, it completely disregards the variations between individual phases. The selection of a single configuration for all the phases could potentially result in suboptimal performance and/or energy-efficiency. This drawback is overcome by the *interphase* tuning plugin, which exploits the dynamism between individual phases by characterizing their behavior. It clusters phases with similar characteristics using DBSCAN [11] and spectral clustering [12] using features such as compute intensity, L2 cache misses and conditional branch instructions. The main motivation driving clustering is that the selected clustering features are capable of depicting the change in the control flow, the amount of work done due to the execution of different algorithms between phases, and the shift from a compute-intensive phase to a memory- or IO-intensive phase.

The *interphase* plugin first uses a random search strategy to evaluate the effect of a randomly selected system configuration from a uniform probability distribution on a single phase in each experiment. During an experiment, the plugin requests for PAPI [13] hardware performance counters and computes features to cluster phases with similar behaviour. To improve the confidence in the tuning result, the plugin performs a targeted or selective tuning of the tuning parameters by probabilistically selecting a system configuration based on the Gaussian distribution. The Gaussian distribution is constructed using the concept of *attractor* configurations, which result in a lower energy consumption, and *repeller* configurations, which result in a higher energy consumption. The idea is to evaluate the next phase with a system configuration that is far from a repeller. The *interphase* tuning plugin ultimately determines the optimal configurations for different clusters of phases as well as rts-specific best configurations for the rts's in each cluster. Thus, the plugin effectively exploits both intra-phase and inter-phase dynamism.

At the end of DTA, rts's that have similar best configurations are grouped into a *scenario* using a *classifier*, and a best configuration for each scenario [14] is determined by the *selector*. This knowledge is encapsulated in a tuning model, which is used by the READEX Runtime Library (RRL) that was developed in the READEX project to dynamically switch between different system configurations for the phase and the rts's during production runs. When RRL encounters an unknown rts that was not seen during DTA, it either switches to the default system configuration provided by the batch system, or performs calibration of the rts using Q-learning [15, 16] if the calibration module is set. If the application exhibits inter-phase dynamism, a runtime cluster prediction library automatically predicts the cluster number of the unseen phases during production runs. The library provides three lightweight predictors, of which, one is based on a second-order Markov chain, and the other two are inspired by one-bit and two-bit dynamic branch predictors. Thus, the cluster prediction library prevents the setting of the default system configuration for an unseen phase, effectively resulting in better dynamic savings.

1.3 Contributions

Our work is an extension of the READEX methodology, with the aim to enable users to leverage inter-phase dynamism, i.e., changing characteristics due to the execution of different algorithms between individual phases to tune the energy-efficiency. The key contributions presented in this thesis are:

- **Extension of the Score-P Online Access interface for call-path profiling:** The performance data measured by Score-P is stored as call-tree profiles that can be accessed using measurement retrieval requests from PTF via the Online Access (OA) interface. The current version of the OA interface of Score-P transfers the objective values and the hardware counters from Score-P to PTF in the form of a flat profile, where the measurements for individual program regions are aggregated regardless of the call-path. Our work extends the OA interface to support the transfer of performance data in the form of call-path profiles, where each rts is stored in a separate node representing one call-path. This is replicated in PTF, thus preserving the parent-child relationship between the calling function and the callee.
- **Tuning plugins to leverage application dynamism:** Our work presents two novel tuning plugins that perform both DVFS and UFS to leverage intra-phase dynamism and inter-phase dynamism respectively. The *intraphase* tuning plugin exploits the intra-phase dynamism arising from the variation in the characteristics such as the compute intensity and execution time across different program regions executed within a single phase. It determines a single best configuration for the phase, and rts-specific best configurations for the rts's of the program regions. However, the *intraphase* plugin disregards the dynamism due to the variation in the execution time between individual phases. The main contribution of our work is the development of a tuning strategy for applications that exhibit inter-phase dynamism using the *interphase* tuning plugin to determine best configurations for groups of similarly behaving phases, and rts-specific best configurations for individual rts's.

  Thus, the *interphase* plugin determines optimal configurations for different executions of the same region depending on the execution context, e.g., interpolation on different grid levels, as well as the data dependent variation, e.g., due to grid refinement resulting in the variation of characteristics of a region over time.

- **A set of metrics to characterize the phase behaviour:** Hardware counters are used to monitor events at the CPU level, and provide detailed insights on the effect of the application execution on the performance of the caches and the main memory. The temporal behaviour of the phases due to the dependence on the type of algorithm that is currently executing can be characterized using PAPI hardware performance counters. In this thesis, we discuss a set of counters that may be used as features to characterize the application using the low-level API for native PAPI events.
- **Search space optimization:** Our approach executes the application, in the worst case, two times during DTA. Moreover, the two tuning plugins limit these runs to a representative set of progress loop iterations instead of the entire application run, thus saving precious tuning time. The *interphase* plugin can select the best configuration for each cluster of phases immediately after clustering in the first run. However, this could potentially result in a suboptimal solution. Hence, we perform search space optimization using a probabilistic random search strategy that uses a Gaussian distribution to select configurations that are closer to the optimum configuration. We also ensure that a previously evaluated configuration is never repeated for a particular cluster, thus increasing the confidence in the tuning result.
- **A runtime cluster prediction library:** Since DTA evaluates and clusters only a representative subset of the application phases, we implemented a runtime prediction library to predict the cluster ids of all the phases that were not seen during DTA. The prediction library implements three types of predictors: a second-order Markov chain based predictor, a one-bit dynamic branch based predictor, and a two-bit dynamic branch based predictor. The application is first prepared for runtime tuning by linking the cluster prediction library. During production runs, the library is initialized at the beginning of the first phase, and collects the PAPI hardware performance counters that were used for clustering during DTA. The cluster predictors then predict the cluster id for the upcoming phase using the features derived from the measured PAPI counters.

We highlight multiple advantages of our methodology. First, we reduce overheads by pre-computing best solutions at design time and simply switching between the configurations at runtime. This reduces the runtime overhead as expensive runtime search techniques are not applied. Second, our approach can be used with minimal user involvement. The pre-analysis steps filter overly fine-granular regions and identify the dynamism automatically. Tuning can be performed immediately by invoking the *intraphase* or *interphase* plugin for an application using automatic compiler instrumentation. The user can additionally improve the overhead due to compiler instrumentation by manually instrumenting the significant program regions. Finally, our work aims to enable developers to achieve significant improvements in the energy-efficiency of HPC applications on extreme-scale systems using an (semi-)automatic autotuning framework that can scale to extreme node counts.

1.4 Structure of the Dissertation

This dissertation proposes software methods in the form of tuning plugins to leverage the hardware innovations of DVFS and UFS to optimize the energy consumption on newer architectures. The remainder of the dissertation is structured as follows:

Chapter 2 describes the related work in power and energy management. It presents the state-of-the-art hardware mechanisms for power management, such as clock gating, power gating, Dynamic Duty Cycle Modulation (DCCM), DVFS, UFS, and power-aware hardware. It then surveys the relevant research works that employ software methods to leverage the hardware innovations for power management, including power-aware job scheduling and combined power-capping techniques.

Chapter 3 introduces the fundamental concepts of our work using a high-level description of the key concepts and definitions that will be referred to in the dissertation. It also gives a background of the existing tools and APIs, such as the Periscope Tuning Framework (PTF), Score-P and PAPI that are the backbone of our methodology. Additionally, it describes the layers of the HPC stack, namely the hardware, system-software and application tuning parameters that our methodology exploits.

Chapter 4 outlines the general overview of the architecture of the existing READEX tool suite, and describes how it leverages two existing tools, namely PTF and Score-P to perform autotuning. It highlights the interactions between the major components of the methodology, and describes the extensions to PTF by presenting the workflow of the *intraphase* and *interphase* tuning plugins. Finally, it outlines the workflow of the RRL to perform dynamic tuning during production runs.

Chapter 5 presents our proposed methodology that exploits the inter-phase dynamism in an application to determine the optimal system configurations. It describes the reasoning behind the features that were selected to cluster the application phases using DBSCAN and spectral clustering. It then describes how the *interphase* tuning plugin selects a targeted set of configurations using a Gaussian probability distribution to determine optimal configurations. Finally, it provides the relevant theoretical concepts and the implementation of the different cluster prediction schemes that are used to predict the phase behaviour at runtime.

Chapter 6 illustrates how our proposed methodology is integrated in the overall tuning methodology. It also describes the domain knowledge that can be specified by the application expert to enhance the tuning process. Finally, it describes the tuning model generation that stores the information of the best configurations determined by the *interphase* and *intraphase* tuning plugins in a JSON file.

Chapter 7 presents four different scientific applications, namely 128.GAPgeofem, sam(oa)^2, INDEED and miniMD that were used to evaluate our tuning methodology. Of these, INDEED, sam(oa)^2 and 128.GAPgeofem are highly dynamic complex real-world applications, while miniMD is a proxy benchmark from the Mantevo benchmark suite. This chapter illustrates the clusters that were obtained for the applications after performing DBSCAN and spectral clustering during DTA, and presents the maximum theoretical static and dynamic energy savings computed using the *interphase* plugin. It also compares the runtime savings achieved using the cluster predictors, and outlines the overheads incurred as a result of runtime switching.

Chapter 8 concludes our work by summarizing the contributions, the challenges encountered, and the lessons learned. Finally, it outlines promising opportunities for future improvements.

2

Related Work

Reducing the energy consumption for Exascale computing has become an important research topic in two main areas in recent years: hardware and architectural support at the circuit and logic level for power- and thermal-aware hardware design, and power-aware software development to leverage the hardware support for the entire software stack [17]. Modern-day hardware designs implement different power management techniques, such as clock gating, clock modulation or Dynamic Duty Cycle Modulation (DDCM), power gating, Dynamic Voltage Frequency Scaling (DVFS), and specialized coprocessors [18] to support the reduction of energy consumption. Software techniques such as Dynamic Concurrency Throttling (DCT), thread/concurrency packing [19], combined with OS-level control are employed to leverage the available hardware innovations.

The energy consumption E is the integral of the instantaneous power consumption of the execution over the execution time T

$$E = \int_0^T P(t)dt \tag{2.1}$$

According to Weste and Harris [20], the power consumption in CMOS circuits comes from two components: dynamic dissipation and static dissipation, and is represented as:

$$P_{total} = \underbrace{\alpha C V_{DD}^2 f + I_{SC} V_{DD}}_{P_{dynamic}} + \underbrace{(I_{leak} + I_{cont})\dot{V}_{DD}}_{P_{static}} \tag{2.2}$$

Dynamic dissipation includes the switching power, $\alpha C V_{DD}^2 f$, resulting from switching of the load, and the short-circuit power. Dynamic power dissipation occurs when the transistors inside a CMOS circuit are switched, allowing internal capacitances to charge/discharge and short-circuit currents to move through the circuit. α, the activity factor is the probability that the circuit node transitions from 0 to 1, which is when the circuit actually consumes power. $P_{dynamic}$ is dominated by the switching power, which is the power consumed while the chip is doing useful work.

Static dissipation occurs due to leakage power due to subthreshold, gate, and junction leakage as well as contention currents (I_{cont}). P_{static} is dominated by the leakage power, $I_{leak}V_{DD}$. Unlike dynamic power, static power depends on the fabrication technology used rather than the switching frequency [21]. This means that the more transistors that are crammed inside a single die, the higher the leakage power. Thus. P_{static} is present even if no switching is performed.

From Equation 2.2, we can see that the source voltage V_{DD}, and the switching frequency f determine the amount of dynamic power. Thus, $P_{dynamic}$ is expected to be smaller during processor idle periods. Modern power-saving techniques target different parts of Equation 2.2 to save either dynamic or static power.

Section 2.1 presents hardware techniques and mechanisms that can be used to reduce the energy consumption of the whole system. Section 2.2 describes in detail the state-of-the-art software techniques including energy/power-aware methods for heterogeneous systems that leverage the hardware and the underlying architectural design for improving power and energy-efficiency. Section 2.2.6 classifies existing works in terms of the control method employed to tune the energy-efficiency or power consumption.

2.1 Hardware Mechanisms for Power Management

Power saving may be considered in terms of active, standby, and sleep modes. The ACPI (Advanced Configuration and Power Interface) standard [22], which is a platform independent interface for configuration and power monitoring and management, defines five main states that are supported by current processors: global states (G-states) [23], system sleep states (S-states), processor power states (C-states), processor performance states (P-states), and throttling states (T-states).

From a high-level perspective, global system state G0 is the working state, while G1 is the inactive or sleep state, and G2 is the shutdown state. During the active state G0, the CPU faces varying levels of utilization, which translates to different C-states. C-states typically range from C0 to C6, with additional vendor-specific states where only C0 is the working state and the higher C-states correspond to deeper and deeper idle states in which no instructions are executed. This means that deeper C-states are low-power modes that can be used when the CPU is idle in order to obtain higher power savings. However, deeper C-states need an external signal, e.g., an interrupt, to return to a working state, and have a higher latency to return to C0, which creates considerable overhead. At C0, multiple P-states associated with an operating frequency and a voltage exist. The number of P-states are dependent on the processor model [23]. P0 operates at the highest operating frequency, and successive higher P-states represent lower clock frequencies, resulting in lower power consumption [22]. T-states are used to perform on-demand clock modulation, which refers to the fraction of time that a system is in an active state, and involves omitting duty cycles.

The following sections describe the hardware mechanisms available for power management. Each of these hardware mechanisms affects different system and processor states.

2.1.1 Clock Gating

Clock gating enables power consumption by disabling the clock in the idle parts of the circuit, or parts that maintain a steady state and do not need to be refreshed [17]. This is done by ANDing a clock signal with an enable signal in order to turn off the clock to idle blocks. For example, when no floating point instructions are being issued, the enable signal is active, and thus, the clock for the floating point unit can be turned off. Clock gating affects the dynamic power $P_{dynamic}$ [20] by influencing the activity factor α in Equation 2.2, since α becomes 0 when the circuit can be turned off. In clock gating, only the clock is disabled while keeping the power supply on, so that the unit retains its state. Since the clock network has a high capacitance, clock gating saves significant power.

Clock gating can be performed at the hardware level at a finer granularity by disabling parts of the functional blocks, and at the software level by disabling entire functional blocks [17]. Clock gating affects the shallow C-states by stopping the processing of instructions [24].

2.1.2 Power Gating

Power gating affects the S-state and deep C-states [24] by reducing the static current during sleep mode by turning off the power supply to sleeping blocks. When a logical block is active, the transistors are switched on, thereby connecting to the voltage supply. During power gating, the power supply is disconnected to the sleeping logical block. Power gating is a more aggressive approach than clock gating, and achieves a better power reduction by reducing both dynamic and static power, since a functional block is disconnected, thereby powering off all its components. Power gating can also be used in combination with clock gating [17].

Power gating requires a number of hardware design decisions: keeping a minimal delay to the circuit during active operation, keeping the leakage low during sleep mode, ensuring that the block is turned off for longer intervals since the transition between active and sleep modes takes time and energy, saving the state of the registers into memory before power gating since the data is erased when the block switched off, and finally, reloading the registers from memory upon power-up [20]. The main concern of power gating is the overhead incurred in time and energy when blocks are powered on and off. Therefore, it is crucial to determine when to use power gating, and whether the overhead is worth it. These decisions are made using a set of strategies known as Dynamic Power Management (DPM) [21, 25].

Dev et al. [26, 27] performed power gating for GPUs, and concluded that it improves overall power-efficiency in GPUs by reducing both dynamic and leakage power in the logic. They also concluded that power gating can be used to boost the frequencies of the active control units to improve the performance under a given power budget.

2.1.3 Dynamic Duty Cycle Modulation (DDCM)

Clock modulation, or Dynamic Duty Cycle Modulation (DDCM) forces a part of the total number of CPU clock cycles, or duty cycles to be idle, and involves the T-states. By gating a fraction of the clock cycles to a core, the frequency of each core can be adjusted nearly instantaneously, and requires less time as compared to other techniques, such as DVFS [28] (see Section 2.1.4). The advantage of DDCM is that different duty cycles can be set for individual clock domains. Gating the duty cycles prevents the clock signal from driving the processor chip for that time period. For example, a T-state with a duty cycle of 75% makes the CPU run at the same clock frequency at the same voltage, but forces the CPU to be idle 25% of the time [22].

Clock modulation was initially intended to prevent processors from overheating [29], but today, power management can also be performed in addition to thermal control. DDCM saves power more effectively for imbalanced applications because the hardware can provide more power to the critical thread [28]. However, it is recommended to use the P-states in modern x86 systems than T-states or even C-states since it is more beneficial in terms of overall system performance and time, as it takes less time to move from C0 to deeper C-states or between different P-states than it does to force down the processor duty cycle [22]. T-states could instead be used as a last line of defense against critical thermal conditions by using a catastrophic shutdown detector to immediately halt the processor execution when the core temperature reaches the temperature limit while the system is in the highest-power P-state. The modulation is stopped once the temperatures return to non-critical levels.

2.1.4 Dynamic Voltage Frequency Scaling (DVFS)

Dynamic Voltage Frequency Scaling (DVFS) is the most commonly used power saving technique at the processor level. Intel introduced the Integrated Voltage Regulator (IVR) in the Haswell processor, which allows individual cores to have their own voltage-frequency domains. This allows the setting of per-core P-states [30]. Most systems have time-varying performance requirements when computation is interspersed with communication. DVFS can be used to set the processor clock frequency to the minimum sufficient value so as to prevent performance degradation, and reduce the supply voltage to the minimum value sufficient to operate at the set frequency, thereby saving large amounts of energy. The controller determines the supply voltage and clock frequency using the information from the system about the workload and the die temperature [20].

DVFS affects both dynamic power and static power due to its effect on the supply voltage V_{DD}. However, since it has a cubic effect on the dynamic power, the effect on the static power is generally ignored. In DVFS, the CPU is not shut down like in power gating, and thus power savings are not as drastic. Since performance has a linear relationship with the frequency [20], using a higher supply voltage or frequency setting causes an increase in the performance at the cost of high power consumption, while a lower voltage-frequency setting reduces the dynamic power consumption usually at the cost of a longer application runtime [18]. Therefore, using DVFS to save power is not trivial, and must be must by performed when periods of lower processor performance is acceptable, such as in memory-bound or IO-bound code regions [31].

Currently, DVFS is supported on various processors provided by different manufacturers. Intel offers SpeedStep in its processors, and AMD offers Cool ‘n’ Quiet and PowerNow! [17] in its CPUs and PowerTune for its GPUs.

2.1.5 Uncore Frequency Scaling (UFS)

The uncore components of a processor are not present in the core, but are essential for core performance. The core contains the execution units, L1 and L2 caches, and branch prediction logic, while the uncore consists of the Last-Level Cache (LLC), memory controllers, on-chip interconnect, and power control logic [32].

Earlier Intel processor architectures like Nehalem and Westmere had a fixed uncore frequency [5]. However, this was inefficient in terms of the power consumption at low system utilizations because the L3 cache was run at a higher voltage-frequency setting than necessary [30]. The next generations, Sandy Bridge and Ivy Bridge saw a improvement in the power-efficiency at low system utilizations since the cores and the uncore were combined into a single voltage-frequency domain, and shared the same frequency for both components. Intel introduced the Uncore Frequency Scaling (UFS) in newer processor architectures starting from Haswell, which supports separate core and uncore frequency domains and provides improved power- and energy-efficiency by allowing users to manipulate core and uncore frequencies independently.

During active periods of workload execution, most of the power dissipation comes from the cores, and smaller amounts from the uncore. When the cores are idle or in deep sleep state, most of the power dissipation comes from the uncore. Previous research has shown that changing the uncore frequency has a significant impact on memory bandwidth and cache-line transfer rates, and can be reduced to save energy [5, 6, 33, 32] by setting the P-states. Hence, UFS has now becomes a topic of significant interest for research in developing methods for improving the energy- and power-efficiency in HPC.

2.1.6 Power-Aware Hardware

Specific processor architectures allow using soft power capping, for example, by running the system at a lower power with a trade-off for performance, setting a Thermal Design Power (TDP) limit by specifying the amount of power that the CPU can consume, or specifying a time window and a maximum average power using RAPL in the Intel Sandy Bridge processors.

According to Jin et al. [31], the energy to move data between the cache and the memory increases proportionally to the bandwidth and the transport distance. Hence, both hardware and software cache reconfiguration techniques, such as turning off cache banks that are not in use when executing applications that are not data-intensive, as well as data locality have been proposed. Since memories have a much lower power density than logic, it is better to use a larger memory than a faster processor [20], and hence today's chips have memories comprising more than half of the entire area.

Another recommendation is to use special-purpose functional units like accelerators, such as GPUs and coprocessors that offload compute-intensive tasks from the processor for graphics and networking applications. The use of hybrid systems (CPU + accelerator) enables both power and performance improvements at runtime, since the offloading of computations can be pre-conditioned based on the constraints [19]. A high-level design technique called Multi-Vdd [21] can be used to divide the system into several voltage domains consisting of different components, such as a CPU or an accelerator, where each component has its own individual voltage supply and clock.

Additionally, the use of efficient parallel computing architectures including low-power microprocessors, such as ARM processors used in smart phones and tablets, FPGAs and heterogeneous systems may be employed in order to save energy [31]. Another way to reduce the power consumption is to operate more cores simultaneously by reducing the supply voltage to slightly above the threshold voltage, called Near-Threshold Voltage (NTV) under a given power budget at the cost of performance [31].

In addition to hardware methods that can support power management, we can opt for a co-designed hardware and software effort by simultaneously developing the hardware, execution model, OS/runtime, and applications using a joint team of computer architects, system software experts, compiler developers, and application experts [34]. One such joint effort is the Runnemede architecture, which is built for energy-efficiency [34] for extreme-scale computing. However, it has limited compatibility with previous operating systems and applications. The architecture uses NTV circuits, and provides support for clock and power gating in processors, memory modules and networks to minimize power dissipation. Runnemede's processor chip includes a large number of relatively simple, single-issue, in-order cores that are organized into hierarchical groups, and divided into Execution Engines and Control Engines to allow separate optimizations of the hardware.

2.2 Software Methods for Power Management

Hardware innovations provide a rich set of techniques, as described in Section 2.1, which enables parallel computing software, including system software and applications to exploit and efficiently manage the energy utilization. Power saving can be achieved at the system level, node level or core level. In addition to exposing hardware features to manage the CPU power consumption, the ACPI provides Operating System directed Power Management (OSPM) [22]. OSPM enables the OS to manage the power by defining states of consumption, and modify the resource behavior depending on the state it is in by lowering the speed of operation when no jobs are executing in a certain time interval.

Setting the tuning parameters manually for different program regions is difficult as it requires application domain knowledge. A solution is to use an automatic optimization approach called *autotuning* to automatically generate a search space of possible combinations of tuning parameters, and evaluate them using experiments [7]. Autotuning techniques explore the search space using different search strategies, such as exhaustive, random and heuristic-based, which are described briefly in Section 6.2.

Autotuning can be performed at compile time, which selects a static-best configuration at design-time in an offline tuning phase, at runtime using online techniques, or a hybrid of runtime execution of a model that is computed at design-time [7]. Autotuning techniques optimize different objectives, such as the execution time, energy consumption, Energy Delay Product (EDP), Energy Delay Product Squared (ED2P), and the Total Cost of Ownership (TCO) depending on the end goal. While many techniques optimize one objective at a time, some methods also perform multi-objective tuning.

Some software techniques select the best configuration using analytical models or predictive models to predict the performance or power/energy metrics for a specific tuning objective using the data obtained in a pre-analysis step, and are described in Section 2.2.3.4. In recent years, a close connection has developed between machine learning and high performance computing, since machine learning algorithms have been employed to perform autotuning for both energy and performance. Section 2.2.3.5 presents these techniques, which learn a correlation between different performance metrics and the tuning parameters or tuning objectives, and predict the optimum configuration for the application.

Software methods also include the following:

- Compiler-based power control using code generation tools that perform code transformations to generate optimized code using compiler optimizations. These optimizations include changing the Instructions Per Cycle (IPC), the sequence of instructions, levels of parallelism [7], or loop-level code optimization using the loop unroll factor [18]. Tiwari et al. [35] tune cache tiling factors, loop unrolling factors, and the clock frequency for minimizing the performance, energy, EDP and ED2P. Their proposed autotuner first performs an offline search using Active Harmony [36], a code generation framework to obtain new code variants for different settings of the tuning parameters, and at the same time monitors the power consumption to avoid performance degradation.
- Dynamic Concurrency Throttling (DCT), where the number of active cores are reduced by restricting the number of OpenMP threads for multi-threaded applications, and in effect, forcing some of the cores into an idle mode due to the reduction in the active concurrency [19, 18].
- Enabling the effective use of transistors by using special instructions, such as SIMD vector instructions for the x86 architecture or the AES (Advanced Encryption Standard) instructions for encryption and decryption [17].
- Using virtualization to reduce power consumption by using a smaller number of more powerful, less power-hungry physical servers to host multiple virtual servers [37].

The following sections present a survey of the state-of-the-art software techniques, such as power-aware job scheduling (Section 2.2.1), DVFS (Section 2.2.3), UFS (Section 2.2.4) as well as power-aware optimizations for heterogeneous systems (Section 2.2.2) that leverage the hardware mechanisms described in Section 2.1 to improve the energy-efficiency and/or power consumption of parallel applications.

2.2.1 Power-Aware Job Scheduling

HPC systems execute multiple jobs simultaneously using the job scheduler, which distributes waiting jobs to available compute nodes. Previously, job schedulers distributed jobs with the aim of improving the performance and maximizing the overall system utilization. However, this is a suboptimal solution for today's supercomputing systems, since saving energy is also a target. Hence, the job scheduler can be used to monitor and control the energy consumed by tracking the energy usage in real time and predicting power requirements, since it has a global view of the system, including the available compute resources, job start and end times, and the performance requirements of waiting jobs [31]. Power management can be divided into fine-grained (DVFS) and coarse-grained strategies. Power-aware allocation is a coarse-grained technique in which the job scheduler allocates sufficient compute nodes to queued jobs based on the power status of the overall system and the power requirement of the job [38].

SLURM (Simple Linux Utility for Resource Management) is an open-source resource and job management system that is used in many HPC systems around the world, and uses the Power Adaptive Scheduling (PAS) algorithm [39] to perform resource monitoring and job scheduling. When a job is submitted, the scheduler predicts the power consumption of the cluster if the job were hypothetically executed by summing up the maximum power consumption of the requested resources. If it is higher than the allowed power budget, the algorithm checks if DVFS is allowed for the job, and then calculates the power consumption of the job with DVFS. If this value lies under the power threshold or power cap, the job will be scheduled for execution, otherwise the job remains in the queue to prevent violations of the power constraint. In cases where resources remain idle, some nodes are turned off, and the available power is redistributed by the scheduler in order to start jobs faster. In the Extended Power Adaptive Scheduling algorithm [40], SLURM has plugins for ARM and Intel architectures dedicated to gathering resource usage information per node for an executing job. The plugin is dynamically updated with the CPU and memory utilization of all tasks on each node to enable the scheduler to make more accurate estimates.

A power-aware job scheduler designed for IBM BG/P [41] uses a variant of the First-Come-First-Serve (FCFS) allocation by first creating a scheduling window consisting of an optimal combination of jobs that can achieve the maximum system utilization, and at the same time, consume less power than the power budget. This selection guarantees both fairness by selecting jobs based the system's original scheduling policy, as well as the power budget requirements. The scheduling algorithm then uses a 0-1 Knapsack model to schedule a subset of jobs into the set of available nodes in the system while maximizing the number of nodes allocated such that their aggregated power consumption is under the power budget.

Auweter et al. [42] developed an accurate prediction model for different application workloads for the LoadLeveler scheduler, and was used on the SuperMUC system at LRZ (Leibniz-Rechenzentrum) to perform energy-aware scheduling. The LoadLeveler provides the users with an energy tag to uniquely identify similar jobs based on the executable and the input parameters. When a new job with an unknown energy tag is run, the LoadLeveler first uses the default frequency and collects hardware performance metrics as well as the execution time, power consumption, and energy consumption of the job. The next time that a job with the same energy tag is submitted, the Loadleveler uses a multiple linear regression model to approximate the power consumption. It then performs DVFS by selecting an appropriate CPU frequency for the application execution by optimizing the trade-off between the energy savings and the execution time.

Conductor [43] performs intelligent power-distribution to nodes and cores at runtime to improve performance. It perform node-level power capping while trying to cut down slack times by using a power/time Pareto-curve to select the best configuration for individual MPI ranks. During the first time step, Conductor assigns an equal amount of power to each MPI process to achieve an optimal application performance for a given RAPL-enforced power-bound. Next, Conductor uses DCT to select the optimum threads. and assigns a thread and frequency configuration for each MPI process in order to explore the search space. It then monitors the application execution during every time step to predict the behavior of upcoming tasks. Conductor finally performs adaptive power balancing by reducing the power consumption of the non-critical parts of the application, and uses the excess power to speed up the critical path.

Although future supercomputers will have more compute nodes, the cost of operating these systems will restrict the power that can be supplied [31], resulting in the limitation on the number of cores that can run at peak performance simultaneously. However, most supercomputers don't fully utilize the maximum allocated power allocated on each compute node. Therefore, *hardware overprovisioning* was proposed by Patki et al. [44] for power-constrained systems, and was shown to improve overall system throughput, and decrease the average turnaround time. A hardware overprovisioned system allocates less power per node and provisions more nodes, thus enabling it to operate under the same power budget. The benefit is that capping the package (CPU) and memory power below peak power results in all of the power being used, and thus enables the system to operate all nodes simultaneously [45]. The drawback is that at lower power bounds, the most efficient processors are efficient, and at higher power bounds, the least efficient processors are efficient. As only a few nodes run at peak power, jobs must be carefully scheduled so as to reach the optimal performance for the execution [46].

Gholkar et al. [46] also argue that a naive approach of enforcing a power constraint for a job by dividing the power budget uniformly across all the processors suffers from performance variation between identical processors. They proposed a two-level variation-aware machine-wide solution consisting of PTune and PPartition for managing power on a hardware overprovisioned machine, and tested their approach on an Ivy Bridge cluster. PPartition partitions a machine's power budget across parallel jobs running on the system such that the power budget is never exceeded. PTune determines the optimal number of processors for a job by eliminating the less power-efficient processors, and distributes the job's power budget across them. This aims to maximize the job's performance under the power budget. PTune uses an offline model to characterize the performance by maximizing the number of retired Instructions Per Second (IPS). It uses RAPL via the msr-safe kernel module to collect power and performance profiles of MPI applications. Once a job is dispatched by the scheduler, PPartition calculates its power budget. If there is not enough power, it steals power from previously scheduled jobs, or only schedules this job once sufficient power is available.

Sarood et al. [45] profiled the strong scaling of an application under different power caps to optimize the number of nodes and the power distribution between CPU and memory subsystems in order to minimize the execution time. Instead of profiling the entire search space of different node configurations, CPU power levels and memory power levels, the proposed method optimizes the search space by estimating the performance using curve fitting or interpolation. Their approach uses RAPL to obtain power readings for each power plane via Machine Specifics Registers (MSRs) to perform power capping for overprovisioned systems. Their experiments were run on Sandy Bridge servers.

Imes et al. [47] present CoPPer, a feedback controller that performs software-defined power capping as a replacement for software-managed DVFS control. It is designed to be system, application and power capping implementation independent, and performs power-aware job allocation while meeting performance targets. CoPPer uses an adaptive feedback control using a soft performance goal, performance feedback, and the minimum and maximum power caps of the system to perform power capping. After each time window, an adaptation function uses the application performance to return the new power cap that should be applied to the system to prevent over-allocation of power when applications cannot achieve an additional speedup. The evaluation was done using RAPL to perform power capping. The authors conclude that the overhead from power capping is lower than the overhead from DVFS, and coarse-grained socket-level power capping provides better runtime benefits over fine-grained DVFS.

Job scheduling techniques for reducing the power consumption may also be applied when application Service Level Agreements (SLAs) for VMs have to be fulfilled. The job scheduling policy must consider many factors, such as consolidating workloads, preserving the QoS of the tasks as agreed on the SLA, and the virtualization overheads incurred during VM creation, checkpointing and migration [37]. The method proposed by Berral et al. [37], introduces such a job scheduling strategy that decides the best location for executing a new job depending on the resources it requires in order to fulfill its SLA. The method periodically calculates whether to move jobs using the information derived from the system, including job execution and node status. When new VMs have to be created, the strategy decides the optimal combination of the hosts or the physical machines to start up or shut down. The strategy takes into consideration the heterogeneity of the machines in the cluster, the hardware and software requirements of the VM, the amount of resources as well as the energy consumption. Using the above constraints as input, the scheduling policy ranks each machine using a dynamic score, and solves a dynamic optimization problem using a hill-climbing algorithm to assign each VM to the best machine.

2.2.2 Power Management for Heterogeneous Systems

As described in Section 2.1.6, power-aware methods are becoming more important for heterogeneous systems consisting of accelerators such as GPUs, especially as HPC is increasingly being used for machine learning and AI algorithms. Heterogeneous systems face a high degree of variability in the power consumption due to the presence of processing elements of different architectures [48], and a major challenge for these platforms is a standard energy measurement and monitoring interface [17].

The research by Tsuzuku et al. [48] leverages the iterative behavior exhibited by scientific applications to perform power capping for nodes with accelerators. The method uses a hybrid static/dynamic model to relate the clock speeds with the power consumption and the execution time in order to determine the initial frequency of the GPUs and CPUs at the beginning of the application run. The power and energy consumption are monitored during the application execution for all GPU and CPU clock combinations. Finally, the best clock combination that does not exceed the power limit is selected, while minimizing the energy consumption. In the dynamic tuning stage, the GPU frequency is adjusted based on the monitored real-time power consumption at application runtime. GPU clock speeds are reduced if the power consumption reaches or exceeds the power limit, or increased if it is under. This method is most useful when an application displays different access patterns.

The ANTAREX project [49] uses a Domain Specific Language (DSL) approach to distribute code between multi-core CPUs and accelerators by specifying adaptivity strategies, parallelization and mapping in the application at design-time. The technique introduces an extra compilation step to translate the DSL into the intended programming language. At runtime, the ANTAREX power manager estimates the energy consumption using an offline linear model [50] that correlates the capacitance with the IPC. The goal of ANTAREX is to optimize the performance under a power constraint by using the priorities for each core and the resource usage of the application to solve an optimization problem and select a set of frequencies for each core. The power manager enforces node-level power capping using msr-safe to change the P-state of each core independently. This approach is specialized for ARM-based multi-cores and accelerators, while our approach targets all conventional HPC systems that support DVFS.

2.2.3 DVFS

DVFS for parallel applications can be applied at coarser granularity by setting a single optimum frequency for the entire application run or for specific workloads, and at finer granularities by setting different frequencies for different application phases. Previous works in phase detection and prediction have multiple definitions for the phase. One definition for program phase is that a phase is a period of execution with a stable behavior [51, 52]. The alternate definition defines a phase as a single iteration of the time loop [53, 54]. Phase identification methods can identify phases at fine- or coarse-granularity. For coarse-grained identification, hardware performance counters are used to dynamically identify and predict workload changes, and are described in detail in Section 2.2.3.3.

Most fine-grained phase classification methods divide the application's execution into non-overlapping, fixed size execution intervals using the number of executed instructions. The phases of the profiled application are then identified as compute-, memory- or IO-intensive using the similarity in the behaviour during individual execution intervals. The characterization of the application execution into similar regions depends on the target application [55]. According to the first definition of a phase, a correctly identified phase would have very small variations between any two execution intervals in that phase. DVFS is then performed by applying a different CPU frequency for each phase [31].

Section 2.2.3.1 presents both reactive phase identification techniques that predict phases at a coarser granularity, i.e., sampling, and predictive phase identification techniques for predicting phases at a finer-granularity. These techniques may not use DVFS specifically. However, the phase identification methods could be considered as a starting point for power-saving techniques. Section 2.2.3.2 presents methods that characterize the application execution into phases, and then also perform DVFS after phase prediction.

2.2.3.1 Phase identification techniques

Sembrant et al. [56] developed ScarPhase, an online phase detection library that collects hardware performance counters using perf events to generate the so-called sparse frequency vectors. The frequency vectors are constructed using perf events that either count the occurrences of certain events, such as the number of instructions executed, or perform sampling by periodically triggering interrupts after a given number of occurrences of the event and then recording the CPU state. The library uses Intel's Precise Event Based Sampling to sample conditional branch instructions and determine the current phase of an application at runtime with less than 2% runtime overhead.

Nagpurkar et al. [57] also developed an online phase detection mechanism using a similarity model to transform execution profile elements into a sequence of similarity values using conditional branches. The similarly values represent the degree of similarity between profile elements. If the similarity is sufficiently high, the model marks the current execution as belonging to a certain phase, or in transition between phases. The framework identifies phase boundaries by correlating a period of repetition with the time of the latest dynamic branch. The drawback of this mechanism is that it assumes that different iterations of a loop and recursive executions of a method belong to the same phase even if they have varying branch behavior.

The work by Kim et al. [58] identifies and predicts program phases using a phase tracking hardware to exploit phase behavior in order to perform dynamic optimizations by adaptively reconfiguring the microarchitecture, such as the cache size. Dynamic optimization systems track phase changes by sampling the performance counters or by code instrumentation. The phase tracking hardware tracks functions and loops in the program using a stack of the path from the root node to a dynamic code region. A phase history table is used to track the change of phase signatures between dynamic code regions. A change in the phase signature is detected using Clocks Per Instruction (CPI), and then assigned a phase ID.

Padmanabha et al. [52] demonstrated that heterogeneous systems can gain energy-efficiency by using a predictive scheduling or switching mechanism that dynamically guides program execution to the most energy-efficient core, without causing a performance degradation. Their work involves identifying phases at a finer granularity and mapping them to customized hardware. They developed a predictive controller to predict an oncoming phase change at a granularity of hundreds of instructions, and accordingly migrate a thread using this information. They argued that their predictive approach is more energy-efficient that existing reactive controllers, which distinguish coarse-grained phase changes by sampling and assume that the performance will remain stable until the following sampling phase. Their predictive scheduler reduces the energy consumption of an out-of-order processor by 15% with negligible overheads on a tightly coupled heterogeneous system.

Gonzalez et al. [59] developed a fine-grained phase identification technique using hardware counters that are read only when MPI communication (collective or point-to-point) takes place so as to characterize the behavior of a single computation or CPU burst that takes place between communications. The technique uses completed instructions and IPC to perform a cluster analysis on the computation bursts. The goal of this technique is to use IPC to detect application regions with different computational complexities, as well as regions with the same complexity but different performance. The clustering is done using DBSCAN, a density-based clustering algorithm in combination with a variant of agglomerative hierarchical clustering.

2.2.3.2 Detection and characterization of application behaviour for DVFS

Zhang et al. [60] present PIFA, an intelligent Phase Identification and Frequency Adjustment framework for energy-efficient and time-sensitive mobile computing by combining offline analysis and online analysis. Their main motivation is that a single frequency setting for different application phases may not be energy-efficient enough. The offline analysis profiles an app to collect the resource utilization, and performs cluster analysis using K-Means to identify the major phases, which are representative of the major energy consumption patterns exhibited by the app. Then, a phase classifier is trained to dynamically identify phases during the online analysis using resource usage information collected from the CPUs, GPUs and memory. At runtime, PIFA schedules an app to execute on an idle CPU core whenever possible, and uses the phase classifier to identify phases and perform DVFS.

Sherwood et al. [61] developed *SimPoint* to find simulation points in a program by clustering the code profile of the full execution, and then picking a point from each cluster as representative of the program execution. The advantage of this approach is that the entire analysis is independent of the architectural parameters, and uses only the code execution characteristics for clustering. It captures information about the changes in the program behavior over time using a Basic Block Vector (BBV), which represents a code section with a single entry and exit. The similarity between different intervals is determined using the frequency of execution of the basic blocks by tracking the program counter of every committed branch and the number of instructions executed between the current branch and the last branch. If the distance between two basic block vectors is small, they are similar. The tool uses random linear projection to reduce the dimensionality of the input data, and clusters similar intervals into phases using K-Means. The drawback of this method is that K-Means requires the number of clusters, k as input, so the tool performs clustering for all possible values of k and selects the best result. The authors extended *SimPoint* with an online phase prediction architecture using a Run Length Encoded (RLE) Markov predictor [51] to perform DVFS to achieve energy savings for a relative reduction in performance.

The Runtime Energy Saving Technology (REST) [62] modifies the core frequencies by performing DVFS at runtime without prior knowledge of the application on two different Xeon architectures. The proposed method characterizes the phases of an application workload into compute-bound or memory-bound states by implementing an interrupt-based sampling of hardware performance counters to determine the state of the processor and chip-level memory activity. The data from the profiles is interpreted by different kinds of decision makers to determine the frequency whenever there is a shift in the workload. The aim is to lower the frequency during a memory phase and raise it during a compute phase. Frequency switching is performed using different decision-makers. The naive decision-maker always changes frequencies when necessary by accepting the profiling data to be true. A second decision maker determines a change in the frequency during workload changes by verifying the profile data using a history table of past profile data. Another decision maker studies past workload changes and predicts future changes by using a Markov-based predictor. While this approach is promising, it is coarse-grained, and does not take into account similarities in the behavior of code regions.

Isci et al. [55] perform online phase predictions using a Global Phase History Table predictor, which is inspired by a branch predictor technique to switch frequencies using DVFS. The predictor characterizes application behavior using the metrics memory bus transactions/micro-op and micro-ops/cycle. The application is divided into coarse-grained phases by setting interrupts after each interval of millions of instructions to prevent any observable overheads. The phase predictor is invoked at each interrupt, and observes the patterns from previous samples to determine the behaviour of the next phase. The prediction is mapped to a DVFS setting, which is then applied to the processor for the next interval using the Intel SpeedStep technology.

Booth et al. [63] propose a Hidden Markov Model framework to detect application phases to apply DVFS, which the authors call Phase-based Voltage and Frequency Scaling (PVFS) to save power for NoCs. Unlike Markov chains that contain only observable states that can visibly be seen or measured, Hidden Markov chains contain hidden states that influence the probability of transitioning from one state to another. The Hidden Markov Models are constructed using hardware counters such as IPC taken at specific code locations, for example, at individual iterations of the time loop. The hidden states are the phases that represent specific power and performance characteristics, such as {high power, low performance}, for which a static DVFS setting is generated. The advantage of this method is that since the Hidden Markov Model is constructed using hardware counters, it can be applied to any heterogeneous many-core system. The limitation is that it statically sets a single frequency for the phases.

Acun et al. [54] propose a fine-grained runtime approach to leverage application dynamism using Charm++ by considering function to function variations within the applications to optimize the energy-efficiency using DVFS. The approach runs only a subset of the total iterations of the application time loop. First, the runtime collects the execution time and the energy consumption for each instance of a region in the application under different frequency settings. Then, optimal frequency is calculated and applied for each instance to either minimize the energy or minimize the performance degradation. The implementation was tested on Haswell processors using the cpufreq kernel module to switch the core frequency. While this approach is similar to our proposed approach, it neglects potential similarities between the individual iterations of the time loop, which our work focuses on.

Intel's open-source Global Extensible Open Power Manager (GEOPM) [64] is a collaborated community effort to develop a tree-based, runtime framework that provides energy management for power-constrained systems. The framework is plugin-based, and supports offline and online analysis. The offline analysis first performs a training run of the application to characterize it, and then collects the energy and performance data to determine the optimal core frequency for individual program regions. At runtime, it collects hardware performance counters from the instrumented application, estimates the power consumption and dynamically identifies and reallocates power to the nodes on the critical path. The goal of GEOPM is to adjust the power caps for the nodes individually instead of performing a uniform power capping. The drawback of this method is that it only switches the core frequency, and ignore similarities between individual phases.

Schöne et al. [65] developed a method to integrate performance analysis with energy-efficiency optimizations by using the profile data of instrumented applications on Intel Sandy Bridge, Westmere and Ivy Bridge systems. They perform region-based DVFS to switch the frequency, and DCT to change the number of threads. First, application profile data is collected along with PAPI hardware performance counters to measure the L3 cache miss rate. The profile data is used in a post-mortem step to define the adaptations that have to be made whenever a particular region is entered. For example, the frequency is lowered when the L3 cache miss rate is over a certain threshold, and increased to the reference frequency when the miss rate is below the threshold. The major drawback of this method is that it assumes that regions exhibit a similar behavior across runs for different input data, which is not necessarily true, as described in Section 8.1.

Hotta et al. [66] implemented PowerWatch, a profiler to measure and collect the power information of each node in a cluster while performing DVFS for individual program regions of an application. The program is divided into several communication- and compute-intensive regions by manually instrumenting the code at appropriate locations. The instrumented code is then executed to collect the profile data for the execution time and power consumption for each region at every frequency setting. Finally, best clock frequency settings are determined by taking the switching overhead into account in order to optimize the EDP and Power Delay Product (PDP). The settings are then applied to individual regions during production runs. The drawback of this method is that the code regions are manually instrumented, while our proposed method also works with automatic compiler instrumentation.

2.2.3.3 Using workload characterization

HPC systems run multiple applications at the same time, where each application can have different resource requirements, and also exhibit variations in the workload at runtime. Hence, it is important for systems running applications with varying workloads to select the voltage-frequency setting dynamically in order to adapt to the workloads [67].

Basireddy [67] developed a runtime manager that identifies changing workloads on the processor and maps a workload to an appropriate DVFS setting using cpufrequtils at runtime. First, the approach selects a resource combination that meets the application performance requirements using a performance prediction model using the metric Memory Reads Per Instruction. At runtime, it monitors the application workload and collects performance data via hardware performance metrics. It then classifies the workload into classes (compute-intensive, memory-intensive, or mixed) using K-Means. The optimal frequency setting is selected either periodically for each core or for the whole system depending on the underlying architecture in order to minimize the overall system energy consumption while maximizing performance. The proposed technique was concluded as a significant improvement over Linux's conservative, ondemand and performance power governors on Odroid-XU3 with ARM's big.LITTLE heterogeneous architecture as well as Intel Xeon E5-2630V2 and Xeon Phi 7120P cores. The drawback of this approach is that the voltage-frequency setting is static for the entire application run.

Chetsa et al. [68] propose an automatic workload detection and characterization methodology to perform dynamic system reconfiguration by using DVFS without the involvement of the user. The method introduces the concept of Execution Vectors (EV) that store sensor values, including hardware performance counters, network bytes sent/received and disk read/write counts and label workloads into compute-intensive, memory-intensive, mixed or I/O types. Changes in consecutive workloads are detected when the distance between their EVs exceeds a threshold, and are used to label recurring phases. DVFS is then performed by switching the frequency depending on the workload type. For example, the disks are sent to sleep and the frequency is scaled up for compute-intensive workloads, and the frequency is scaled down for memory-intensive workloads. The authors extended their work with modeling techniques [69] to understand the runtime behaviour due to changing workloads on other HPC systems. Their prediction model uses the offline data from the hardware counters on the reference platform to perform an online matching of the data on the target platform in order to estimate the energy consumption of the application.

2.2.3.4 Using analytical models

The AutoTune project [8] extended Periscope [70], an automatic performance analysis tool with autotuning support for tuning the performance and energy-efficiency of HPC applications using tuning plugins. The parallelism capping plugin improves the energy consumption of OpenMP applications by withdrawing compute resources whenever they are not utilized efficiently, and uses EDP as the objective function to determine the optimal number of threads for each OpenMP parallel region. The DVFS plugin [71] can tune different objectives, such as the energy consumption, TCO, and EDP by generating models to predict the energy consumption, execution time and power at different CPU frequencies using the performance data on a certain platform. The plugin uses PAPI hardware counters to collect performance measurements, and the enopt library [72] to sample the energy measurements via RAPL every 10 seconds. The plugin also performs a search space reduction using the selected frequency by evaluating the next lower and higher frequency values. This is a static tuning approach where a single CPU frequency is set for a region during the entire program run. In our approach, both core and uncore frequencies can be configured dynamically for different instances of the program regions.

Springer et al. [73] propose a method that explores the energy-time trade-off by developing a model-based algorithm that selects the number of nodes and the CPU frequency to perform dynamic switching for computation and memory phases. The application is first divided into compute- and memory-intensive phases, and then performance models for both execution time and energy consumption are created. The performance is then predicted using estimates that are generated by repeatedly executing the program for a few iterations and performing regressions until a satisfactory configuration is found. The method was tested on an AMD Athlon-64 cluster using sysfs to switch the frequency.

Green Queue [74] uses an application's characteristics as inputs to a power-performance model to perform DVFS in order to minimize the energy consumption of program regions. The framework first uses static properties from binary instrumentation, and runtime behavior from traces to determine whether the application is load-balanced across MPI ranks. It then performs phase characterization, where each phase is said to be a contiguous execution of code whose optimal frequency is approximately homogeneous. Two strategies were implemented for selecting the clock frequency depending on whether the application is load-balanced or not. If the workload is balanced, DVFS is used to switch the CPU frequency for different phases of the time loop. For unbalanced loads, the CPU frequency is varied for individual process ranks. Once the optimum frequency is selected for the minimum energy consumption, neighboring phases with the same frequency are merged. At runtime, the optimal configurations are read from a file and applied to different application regions.

Elangovan et al. [75] propose DVFS to select different frequencies for different memory channels to selectively lower the frequency of the main memory and reduce the energy consumption while keeping the performance degradation within tolerable limits. The algorithm first reads all the hardware counters, and uses a performance model to calculate the execution time as well as the power consumption of certain parts of the program at all the possible voltage and frequency levels. It then chooses a state for the next code region that maximizes energy savings while maintaining the performance. Since the complexity of the algorithm increases exponentially with the number of memory channels, the authors propose various frequency selection strategies, including an exhaustive search in which all the frequencies are considered, a rule-based search where only three frequency levels are considered at a time, and a ganged search by slaving all the memory channels together.

Hsu et al. [76] implemented a power-aware algorithm for an adaptive runtime system that is based on CPU utilization on laptops. The algorithm lowers the CPU voltage and frequency setting when the CPU utilization is low to conserve energy, and vice-versa when the utilization exceeds the threshold. The aim of this approach is that it is both application- and input-independent, so it does not require any profiling information a priori. Therefore, it implicitly gathers information by monitoring the level of off-chip accesses during each interval, and uses hardware performance counters like the MIPS (Millions of Instructions Per Second) rate to model the execution time to make scheduling decisions. When the off-chip accesses are high, the cpufreq module is used to lower the CPU frequency setting without affecting the performance. The drawback of this approach is that it is too coarse-grained due the default measurement interval of 1 second.

Rauber et al. [77] developed an application-specific and an application-independent analytical model to select the optimal number of threads, and minimize the energy consumption and EDP by controlling the CPU frequency using DVFS. They used RAPL to obtain power measurements on Haswell processors. PPEP [78] is a software tool that models the power consumption by reading the hardware performance counters of the cores at regular intervals and predicts the program execution at different voltage-frequency states for AMD processors. The tool uses a regression model that takes nine hardware performance counters as inputs, and uses the metric Cycles Per Instruction (CPI) to estimate the chip dynamic power at different voltage-frequency states.

PuPPET (Power Performance PETri net) [38] is a modeling tool that implements a combined power-aware job allocation and DVFS strategy. The tool allows users to analyze power-performance trade-offs by modeling the dynamic execution of a parallel job using an approximation of the energy consumption via petri-nets. It implements a power-aware job allocation strategy by estimating the power requirement for the job at the head of the queue. If its power requirement is less than the power cap, it is allocated resources. Otherwise, either less power-hungry jobs are executed, or DVFS is implemented by running the processors at a low power state to respect the power cap. The tool was evaluated on Mira, an IBM Blue Gene/Q machine.

2.2.3.5 Using machine learning

Vazquez et al. [79] use application dynamism to apply machine learning to reduce the overhead of design space exploration while finding the best configuration of the total size, associativity, and line size for the L1 instruction and data caches. First, Principal Component Analysis (PCA) is used to perform the feature reduction of the execution statistics of the application's time loop. Then, the execution statistics are taken as input by Artificial Neural Networks (ANNs), and a best cache configuration for each unique time loop is predicted. The predicted configurations are stored and retrieval later when the loop is executed again. This method has the advantage that it can be applied to any type of computing system with configurable hardware components. The methodology achieves an average energy degradation of less than 5% using the predicted configuration.

Cochran et al. [80] present a two-step dynamic energy management technique that trains a multinomial logistic regression model at design-time using a set of performance counters, per-core temperatures, current DVFS setting and thread count as inputs to minimize the EDP and ED2P. It finds the optimal configuration of the threads and the DVFS setting for multi-threaded applications based on the workload characteristics. At runtime, the model calculates a priori the probability of each candidate operating point being optimal for the objective function, and selects the output with the highest probability. The authors also present *Pack & Cap* [81], which again uses a multinomial logistic regression classifier to predict the optimal DVFS and thread packing settings. However, it tries to maximize the performance within a power budget. The classifier is trained offline for each workload to estimate the probability of a combination of frequency setting and thread packing yielding the optimal workload performance within a power constraint. The proposed approach uses the cpufreq governor to perform frequency switching.

Ozer et a. [82] optimize the energy consumption of the HPC systems at the Leibnitz-Rechenzentrum (LRZ) using machine learning techniques to perform DVFS. They use historical data generated from GEOPM to train different regression models to predict the optimum frequency settings for the next timestamp given the information from the previous timestamps. A supervised learning approach such as a random forest is trained offline a priori using the trace history of an application as input to predict the future values of the CPU power and the number of retired instructions. The approach also uses linear regression to make runtime predictions of the optimal frequency settings for different program regions.

Hajiamini et al. [83] propose a Markov-based DVFS method to predict core utilizations and CPU frequency settings based on the execution times between application time intervals. The DVFS setting is modeled as a Hidden Markov Model (HMM), which predicts the next state based on some hidden states/variables that influence the probability of the selection using the Viterbi algorithm. This method uses core utilizations and execution times as the states of the HMM. The probabilities for transitioning from one state to another are obtained by profiling the core utilizations. The algorithm uses K-means to partition core utilization values and the execution time into clusters representing a range of core utilizations (low, medium or high), and execution times. Thus, lower frequency settings are applied to clusters with shorter execution times, since they represent an underutilized core.

Benedict et al. [84] propose an energy prediction mechanism using a bagging tree-based Random Forest Modeling (RFM) approach for OpenMP applications. RFM is a tree-based modeling technique that is used to reduce the variance of an estimated predictor by using an ensemble learning method. The mechanism finds the optimal problem size, code variant as well as the best frequency for different code regions to optimize the energy consumption while maintaining the performance of the code. The optimizer first submits a list of proposed energy optimal configurations to the RFM-based energy predictor, which splits them into training and testing datasets. The predictor then creates a regression model using various metrics like the energy consumption, execution time, cache misses, and total number of instructions that are collected during the execution of the configurations in the training dataset. Using the collected metrics, the RFM predicts the energy consumption of the test dataset. The technique uses RAPL to obtain the energy values.

Reinforcement learning is becoming an increasingly popular technique for autotuning HPC applications that have large complex problem spaces. In reinforcement learning, an agent must learn a certain behavior by a continuous process of trial-and-error interactions with a dynamic environment through punishments and rewards [85]. Reinforcement-learning problems may either use the strategy of finding a behaviour in the search space that performs well in a certain environment, or use statistical techniques and dynamic programming to assess the incentive of taking certain actions. The Q-learning approach is a dynamic programming technique in which the agent observes its current state, issues an action and then observes the resulting new state. A penalty is issued to the agent based on the value of the state transition. The agent finally learns the best state by trying all actions in all states to minimize long-term penalties [25].

Gocht et al. [16] used Q-learning to select the best CPU and uncore frequency settings for different instances of program regions, i.e., call-paths of instrumented functions. First, the initial energy is measured when a region is entered for the first time. The algorithm then searches for new states that decrease the normalized energy consumption. If a new state results in a higher normalized energy consumption, the algorithm returns to the previous state that generated a positive reward. Although the proposed method is region-based, it does not take into account the similarities between the regions in different time loops, and assumes that there is no dynamism between individual iterations of the time loop.

Shen et al. [25] present a system-level dynamic power management method using Q-learning for peripheral devices. The method uses a two-level control model that optimizes the power-performance trade-off so that it operates the system at a relatively constant performance or power to minimize the energy dissipation. The state space for the algorithm is the set of available power modes. The Q-learning algorithm performs CPU power management by controlling the DVFS settings when the compute and memory intensity vary for different workloads. First, the power manager learns the pattern of the workload by keeping the device active during the idle period. When the workload changes, the power manager learns again and puts the device to sleep during long idle periods. This method sets a single optimal DVFS setting for a specific input using the cpufreq driver using the Intel Enhanced SpeedStep technology.

2.2.3.6 Intertask DVFS for applications with load imbalance

Some DVFS techniques are specialized for MPI applications that exhibit load imbalance. These schemes optimize the energy consumed during a slack period by identifying the load imbalance in the application caused due to blocking MPI communication [31]. Then, they selectively switch the clock frequency to ensure that the critical path of the execution is never slowed down [86].

Rountree et al. [86] propose a runtime DVFS system called Adagio that reduces the CPU frequency during communication phases using the sysfs interface. The technique first creates tasks, which are defined as a period of computation between two MPI calls. To predict the upcoming communication calls, a unique signature is created for each task based on the hash of the task that was previously found to succeed the current task. Adagio performs an online monitoring of PAPI hardware performance counters to estimate how fast a task would run if it ran at the fastest frequency. After a task completes, Adagio collects the execution time and determines the frequency schedule for the next execution of that task. The selected ideal frequency is defined as the lowest CPU frequency at which a given task can be run without incurring any slack or slowdown. The disadvantage of this method is that similar communication patterns between HPC applications are not considered during predictions, thereby causing a performance penalty. Moreover, Adagio is applicable only to single-threaded programs.

Lim et al. [87] present an adaptive technique for performing DVFS in communication phases by automatically detecting and identifying communication regions by monitoring MPI calls without any user involvement. The system comprises a training system and a frequency shifting system. The training system consists of a region-finding algorithm, which works by maintaining begin and end flags for each MPI call. When the begin flag is encountered for an MPI call, a new region begins, and hardware performance counters are collected. Using the metric micro-operations/second as an indicator of CPU load, a frequency is selected for regions with low micro-operations to minimize the EDP. When the end flag is encountered, the frequency is reset. The experiments were conducted on an AMD Athlon-64 cluster using the sysfs interface for shifting the frequency. The drawback of this work is that since the frequency switching is done on the fly, it needs coarse-granular MPI regions to measure the hardware counters.

Freeh et al. [88] propose a framework to perform DVFS for MPI programs dynamically. The framework identifies phases of compute- and memory-intensive regions in an application using the idle periods caused when the CPU waits for memory or IO. Phase boundaries are identified when there is a change in the memory pressure, represented using the metric operations/miss. In each experiment, a heuristic executes a new phase using a slower frequency than the one used for the previous phase. If the energy-time trade-off for the new phase is better than the current solution, it is accepted. Otherwise, the algorithm recursively executes the phase with the next lower frequency until the new trade-off is worse than the current, or when all the frequencies are tested. The drawback of this method is that it is not applicable for dynamic applications since it assumes that HPC applications have predictive phases.

2.2.4 UFS

André et al. [89] developed a daemon to automatically switch the uncore frequency and at the same time limit the performance degradation. The daemon uses the ratio of the FLOPs and the memory bandwidth to compute the arithmetic intensity and characterize the application into compute- and communication-intensive regions. The daemon determines how the switching decision impacts the FLOPS and memory bandwidth, and reduces the uncore frequency if the FLOPS drop when compared to the previous iteration, but the memory bandwidth remains stable. The approach used LIKWID to read the counters provided by RAPL and set the uncore frequency. The experiments showed that decreasing the uncore frequency has a negative impact on the latency and bandwidth of the L3 cache and main memory, and no impact on the L1 and L2 caches. The authors also evaluated the potential of UFS for energy savings, and recommend limiting the performance slowdown during UFS to up to 10% in order to reduce the overall energy consumption of the socket and main memory.

Gholkar et al. [90] propose Uncore Power Scavenger (UPSCavenger), a lightweight library to dynamically detect phase changes and automatically set the best uncore frequency for individual phases at runtime to save power while limiting the performance degradation. UPSCavenger detects phase changes by periodically measuring the IPC and socket power using RAPL for DRAM power measurement at every invocation of the phase. The uncore frequency is then switched to the highest value during a phase transition using the msr-safe kernel module. A transition from a compute-intensive phase to a memory-intensive phase is identified as a rise in the DRAM power and a reduction in the IPC. The authors observe that UFS results in reduced power for cache-bound applications, performance degradation for memory-bound applications, and reduced package power for CPU-bound applications.

Both the above methods perform sampling to obtain measurements for the DRAM power. The drawback of both techniques is that they are too coarse-grained due to the large size of the sampling interval.

Gupta et al. [32] demonstrate the efficacy of UFS for heterogeneous workloads on an experimental, heterogeneous multi-core platform consisting of both high performing and low performing cores using a set of real-world applications that are typically run on client devices. The workloads are classified as intermittent, sustained, or multi-threaded. For intermittent workloads, heavy processing is performed on client devices in short bursts, thus generating high IPC counts during activities such as web-browsing. Sustained workloads have sustained levels of higher IPC counts, and are marked by longer periods of activity, such as gaming and video editing. Multi-threaded workloads use multiple threads to increase the parallelism for activities such as media decoding and rendering. Each workload is first evaluated on big cores, and then on the small cores, during which metrics for the application performance, IPC and LLC accesses are collected. The metrics are then used to model both the core and the uncore power.

2.2.5 Clock Modulation

Bhalachandra et al. [28] used a model to inspect the local system state for variables like the time spent in collectives to slow down non-critical threads without impacting the performance. Since a thread performing more work runs at a higher duty cycle than the ones doing less work, the model chooses the lowest clock rate for the fastest threads so that they reach the collective without causing any delay. The model also increases the clock rate for slow threads to prevent them from being last to the collective. For generic loads with imbalance, it was observed that clock modulation saved significant power and energy with minimal impact on performance.

Schöne et al. [24] performed a detailed study of software controlled clock modulation for different processor generations, and concluded that it is a good optimization technique for synchronization calls like MPI_Barrier. They developed optimization models for the performance and energy consumption for the Haswell-EP architecture, and measured the influence of clock modulation on the power consumption and application performance. They also compared clock modulation with DVFS, and observed that for the Sandy Bridge and Ivy Bridge architectures, the processor uses DVFS instead of clock modulation when the clock modulation setting is not high enough, or is the same for all the cores. They also report that clock modulation was used in addition to DVFS only when the clock modulation setting is lower than the lowest supported frequency. The authors suggest using clock modulation when synchronization calls have a low runtime, and DVFS for longer running synchronization steps.

Cicotti et al. [91] developed EfficientSpeed, a library that utilizes both clock modulation and DVFS by providing customizable energy optimization policies to improve the energy-efficiency while preserving performance. The library allows programmers to indicate loops within their application that are good candidates for energy optimizations. The library first runs the instrumented application using the default settings to obtain baselines for the specified loops. It then adjusts the CPU speed to minimize a user-specified objective, such as performance, power, energy or EDP. For each compute phase, the speed of the core is selected based on the selected policy, and performance metrics are collected to adjust the speed for the phase when it is run again.

2.2.6 Classification of Software Techniques

We now categorize existing power and energy optimization software techniques from Section 2.2 that leverage the hardware methods from Section 2.1 on the basis of the tuning methodology employed. We distinguish existing works in terms of their search strategy for search space exploration, the time when tuning is actually performed, the granularity of hardware control that was employed, and finally, the granularity of application control that was exploited by the tuning parameters. The classification is presented in Table 2.1, and defined below:

1. **Search space exploration strategy:** For large search spaces, autotuning techniques may use search strategies to execute a subset of the total iterations or use historical data to train runtime models. Search algorithms may be either model-free or model-based [7]. Model-free algorithms consist of global and local search strategies, and do not use models to navigate the search space to determine optimal settings. Global search strategies like genetic search and evolutionary algorithms find the global best configuration at the expense of a long search time. Local search algorithms like hill-climbing do not perform a detailed exploration, and instead move from a current configuration to a nearby improving configuration within its neighborhood, which is defined by the user or the algorithm. They terminate when they reach a time bound or when no better configuration is found for the current configuration. However, they only find the local minima.

   Model-based algorithms are used to avoid the cost of walking through the entire search space. Instead, they use analytical or predictive models for predicting performance or power metrics. Analytical models usually use statistical models to draw the relationship between different variables. Predictive models may various machine learning models such as supervised learning algorithms that understand and learn patterns in the data, and make predictions during the application run.

2. **Time of tuning:** This category refers to when the tuning is actual performed: offline or online. In offline tuning, usually statistical or performance models are computed when the relationship between the tuning parameters and the tuning objective is known [92]. It can also be performed by generating profiles or traces for the application execution, which are used post-mortem to perform performance or power analysis and tuning. Tuning adjustments are then made between successive application executions [35]. Offline tuning strategies may also use previously tuned code regions to perform a lookup to obtain the optimal configurations. When a previously executed optimal configuration is unavailable, a prediction model is used to find and store the optimal setting offline for future use [93]. In online tuning, the actual tuning is deferred to application runtime, where machine learning is used to create predictive models using the metrics collected during application runtime or from a model derived from an offline step.

3. **Granularity of hardware control:** Power control mechanisms can tune hardware tuning knobs at different granularities, for example, per-core, per-node, or the whole system. At coarser granularity, power capping and power management techniques for parallel applications can be applied at the node or socket-level, and at finer granularity at the core-level. For example, some power-aware job allocation strategies may perform system-level power capping thus enabling coarse-grained control. RAPL provides socket-level power capping capabilities while DVFS allows for a finer-grained per-core frequency control for the Haswell architecture. We categorize previous works into system-level, node-level, socket-level, and core-level power control techniques based on the granularity of the hardware that is exploited.

4. **Granularity of application control for tuning parameter switching:** Optimal power/energy reduction configurations for parallel applications can be applied at coarse or fine application granularity,

Table 2.1: Categorization of previous works on power-aware and energy optimization techniques into (a) Model-free and model-based search space exploration strategy, (b) Offline and online tuning (c) System-, node-, socket- and core-level hardware control, (d) Application/workload static tuning and dynamic tuning of applications.

Level	Category	Previous works
Search space exploration strategy	Model-free	[41], [43], [37], [60], [61], [51], [63], [54], [64], [65], [66], [68], [71], [74], [75], [83], [16], [25], [86], [87], [88], [89], [90], [35], [91]
	Model-based	[42], [46], [45], [47], [48], [49], [50], -[55], [67], [69], [71], [73], [75], [76], [77], [78], [80], [82], [84], [90], [28], [24], [38]
Time of tuning	Offline tuning	[39], [42], [49], [50], [60], [61], [63], [54], [64], [65], [66], [68], [71], [73], [74], [74], [74], [75], [77], [78], [82], [83], [87], [90], [24], [35], [38], [91]
	Online tuning	[40], [43], [47], [48], [60], [51], [55], [64], [67], [69], [76], [80], [16], [25], [84], [86], [88], [89], [90], [28]
Granularity of hardware control	System-level	[46], [45], [47], [37], [67], [68], [69], [25], [39], [41], [42]
	Node-level	[43], [48], [71], [73], [80], [88]
	Socket-level	[47], [89]
	Core-level	[43], [49], [50], [60], [61], [51], [55], [63], [54], [65], [66], [67], [71], [71], [74], [75], [76], [77], [78], [82], [83], [16], [86], [87], [90], [28], [24], [91], [38], [35], [64]
Granularity of application control for tuning parameter switching	Application/workload static	[39], [41], [42], [46], [45], [37], [63], [67], [68], [69], [73], [77], [78], [80], [25], [84], [90], [35], [38]
	Dynamic	[43], [47], [48], [49], [50], [60], [61], [51], -[55], [54], [64], [65], [66], [71], [74], [74], [75], [76], [82], [83], [16], [86], [87], [88], [89], [90], [28], [24], [91]

as described in Sections 2.2.3.2, 2.2.3.6, and 2.2.3.3. For example, while performing DVFS, some methods statically set a single optimum static configuration for the entire application, or in some cases for specific workloads at a coarse-grained level. Application-static configurations are configured at the start of the application run, and are not changed for entire application run. While this technique may result in energy savings, it does not consider the inherent dynamism in HPC applications, and assumes that most applications have predictable phase behavior. As our proposed method describes in Chapter 7, there are many applications that require an analysis of the similarities between individual iterations of the time loop.

At finer granularities, tuning strategies may set different frequencies for different application phases characterized by the level of compute-, memory- or IO-intensity, or occasionally even for individual time loop iterations or instances of program regions. Such dynamic tuning techniques select optimum configurations for different code regions of the application.

2.3 Summary

In this chapter, we presented the state-of-the-art in power management and power-aware optimization methods. Currently, approaches such as hardware overprovisioning, power-aware job allocation, power gating, clock modulation, clock gating, power capping, DVFS, and UFS are used to reduce the energy and power consumption on HPC systems. We also presented software methods that leverage architectural innovations in power and energy management. While our work aims to tune the energy efficiency, other objectives such as execution time, EDP, ED2P and TCO can also be tuned.

Power saving may be considered in terms of active, standby, and sleep modes. The ACPI standard defines five main states that are supported by current processors: global states (G-states), system sleep states (S-states), processor power states (C-states), processor performance states (P-states), and throttling states (T-states). From previous works, we can conclude that many techniques use DVFS for optimizing both energy and power consumption. This may be due to the fact that changing the P-states has a low latency than other states like deep C-states and T-states.

We also note that the definition of a phase varies across across previous works dealing with phase identification and application characterization. While some define a phase as a period of program execution with a stable behaviour, others, including our approach interpret it as a single iteration of the time loop.

Our proposed approach includes methods from a wide range of research areas, such as DVFS, UFS and DCT, application dynamism exploitation using domain-knowledge specification, phase characterization using clustering, multi-step autotuning using a probabilistic search strategy, and runtime prediction of phase behavior. The approaches that were presented in this chapter implement one or more of the above methods. However, we are not aware of existing research that covers all these areas collectively, and at the same time has the potential to scale to future Exaflop machines.

3

Background

This chapter presents the fundamental concepts of our work by introducing a formal description of the concepts and definitions that will be referred to in the following chapters [1]. In Section 3.1, we present the concepts of our methodology, and introduce inter-phase dynamism, which is central to our work. We also describe how the base concepts are applied to define the *tuning potential* metric to quantify the potential improvement that can be achieved through tuning in Section 3.1.3. Finally, we define the tuning model in Section 3.1.3.1, which is the outcome of the Design-Time Analysis (DTA) stage (see Section 6.2), and is used for runtime tuning.

In Section 3.2, we describe two existing software tools that our work leverages, namely the Periscope Tuning Framework (PTF) and the Score-P instrumentation and measurement infrastructure. Additionally, we provide a short introduction to the PAPI framework that we use to collect hardware performance metrics. In addition to a high-level description of the tool infrastructures, we distinguish and describe the tuning parameters of the HPC stack, namely hardware, system software and the application tuning parameters that are used in our autotuning methodology in Section 3.3. Understanding these concepts is necessary to understand the contributions of this thesis.

3.1 Concepts and Formalism

This section introduces the core concepts that will be referred to throughout the remainder of this thesis.

3.1.1 System Structure

3.1.1.1 Regions

A *region* is an arbitrary code in the application, and includes functions, parallel OpenMP regions, and MPI operations. The set of all instrumented regions in the program is denoted by R_{instr}, and comprises all the

[1]This section contains definitions from [53]

program regions that are instrumented and thus visible to the tuning system at application runtime. However, R_{instr} could potentially contain a large number of fine-granular regions, so we reduce the set of regions to the set of *significant regions* $R_{sig} \subseteq R_{instr}$ by selecting regions $r \in R_{instr}$ that cover a significant part of the execution time. This means that significant regions are suitable targets for dynamic switching of the tuning parameters. We shall henceforth use R instead of R_{sig} to denote the set of significant regions.

A *phase region* is a program region that defines the phases of an execution, and is typically the main time-stepping loop of a simulation. Thus, all significant regions should be nested within this region. We assume that there is only one phase region defined in the application.

3.1.1.2 Identifiers

An *identifier* is an element that contains information to identify the current execution or predict the characteristics of the consequent execution.

The set of identifiers $ID_r \subseteq ID$ for a region $r \in R$ consists of the *region name*, *region call-path*, and *region parameters*. The region call-path is the sequence of region instances from the root node to the called region. Additionally, the user can provide region parameters for application regions to characterize their behaviour, *phase identifiers* for the phase region to group phases with similar characteristics, and *input identifiers* for the application input to differentiate different performance and energy characteristics.

3.1.1.3 Runtime situations

A *runtime situation*, or an *rts* is an instance of a significant region, and is denoted by $r_{rts} \in R$ during an execution. An rts can be identified using the region name and the call-path.

3.1.1.4 Phase

In Figure 1.1, we see that the runtimes of different iterations of the time loop of INDEED are not uniform throughout the application run, and thus create application dynamism. To leverage this dynamism, we introduce the concept of a *phase* to characterize the application on account of changing computational characteristics of the rts's across individual iterations due to changing workload distribution.

A phase $ph \in PH$ is an instance of a phase region and thus an rts at the phase level. It is a single execution of the phase region, or an iteration of the time loop. The phases are executed collectively by all of the processes of an MPI application, and thus, all the processes go through the same sequence of phases during the application execution.

3.1.2 System Tuning

3.1.2.1 Tuning parameters

A *tuning parameter* $tp \in TP$ is a tuning knob or a parameter of the HPC stack, e.g. CPU frequency, number of OpenMP threads, accelerator offloading switch, application-level parameters, etc. We focus on the tuning parameters that can be switched at runtime, and also have the potential to influence the energy consumption of an application running on an extreme-scale system. We describe the parameters that were used in our proposed methodology in Section 3.3.

3.1.2.2 System configurations

A *system configuration* $cfg \in CFG$ describes the set of tuning parameters and their associated values. System configurations are switched during the execution of the rts at application runtime whenever the monitoring library enters and exits a significant region.

3.1.2.3 Objective function

The *objective function* is a tuning aspect that determines the best system configuration, and maps an rts and a system configuration onto a real number, and is defined as $o : RTS \times CFG \longrightarrow \mathbb{R}$. The *objective* of tuning is to minimize or maximize a given objective function, for example, execution time, energy consumption, or EDP by varying the system configuration.

3.1.3 Tuning Potential and Dynamism

Tuning dynamism exists in an application if two rts's in the application execution exhibit varying computational characteristics such that different optimal configurations can be applied for the respective rts's, thus making it beneficial to perform tuning. At runtime, the configurations for the respective rts's are switched to improve the objective function.

The presence of application *intra-phase dynamism* indicates changing characteristics of significant regions within a single phase due to the execution of different algorithms within the phase. This essentially means that the rts's of different significant regions will have different best configurations. *Inter-phase dynamism* results from changes in the execution characteristics between different phases of the application due to the change in the control flow while the simulation is progressing through the sequence of phases. The variations in the application characteristics are computed using dynamism metrics, as described in Section 6.1.4.

To successfully apply the above formalism, we need to first estimate the benefit of dynamically tuning an application by determining its *tuning potential*. Since we know that tuning-relevant dynamism exists, the tuning potential quantifies the improvement in the objective function compared to a static system-wide default configuration selected by the batch system at application runtime. The tuning potential of the whole application is the sum of the tuning potential of the individual rts's.

The tuning potential cannot be determined theoretically, and can only be estimated through measurements by leveraging the varying characteristics of different regions and taking the switching overhead into account.

3.1.3.1 Tuning model

A *tuning model* is produced as a result of Design-Time Analysis (DTA). It captures the knowledge about the optimum system configurations for individual rts's in the presence of application intra-phase dynamism as well as for phases in the presence of inter-phase dynamism.

In order to reduce the number of configurations in the tuning model, multiple rts's are grouped into a *scenario* using a *classifier* if they have the same best-found configuration or if they have the same set of values for the identifiers.

A *selector* of a scenario returns the single best configuration with respect to the chosen objective. Selectors can either simply return a single best configuration determined at design-time, or use a cost function as described in Section 6.2.3 to choose from a set of configurations when rts's with different best configurations resulting for different input sets are merged together.

3.2 Tools

Section 3.2.1 describes Score-P, a common instrumentation and measurement infrastructure for DTA and Runtime Application Tuning (RAT). Section 3.2.2 presents the Periscope Tuning Framework, which forms the basis of our work, and was extended using tuning plugins for intra-phase and inter-phase dynamism. Section 3.2.3 introduces PAPI, a low-level API to collect hardware performance counters.

3.2.1 Score-P

Score-P [10] is a joint project driven by leading application performance tools experts, and funded by the German Ministry for Research and Education (BMBF) and the U.S. Department of Energy (DoE). It provides an advanced instrumentation and runtime measurement infrastructure that allows the analysis of the behavior of high performance codes and the detection of performance problems and bottlenecks, and provides information on potential improvements. Through its flexible design, Score-P offers different strategies for application instrumentation, e.g., compiler-based and manual instrumentation. The process of performance analysis in Score-P begins with the instrumentation of the original application by preparing it for the collection of the performance properties. This is done by recompiling the application using the Score-P instrumentation prefix in the original compile and link commands through which performance properties are collected at program runtime. Score-P also provides macros to annotate specific parts of the code that that might be potentially interesting for tuning. We describe this process in detail in Section 6.1.3.

In the traditional approach, the application has to be instrumented for each analysis tool separately, since every tool has its own instrumentation system. To overcome this challenge, Score-P offers a common Opari2 instrumenter for OpenMP programs that serves as the basis for many HPC analysis and tuning tools like PTF, CUBE, Scalasca, Vampir and TAU. The runtime measurement infrastructure of Score-P supports various means of data processing and storage, including summarization and storage as profiles in the CUBE4 format in *cubex* files and used by CUBE, TAU and Scalasca, trace data generation through the Open Trace Format 2 (OTF2) used by Vampir and Scalasca, and an Online Access (OA) interface for direct performance analysis and autotuning at runtime used by PTF. The framework supports a wide range of parallelization paradigms including MPI, OpenMP and CUDA, advanced measurement techniques such as phase, dynamic region and parameter-based profiling, sampling through interrupts as an alternative to direct instrumentation, and recording hardware performance counters via PAPI [13] and perf.

The current design of Score-P returns performance metrics for individual programs using region-based profiling via the OA interface. We extended the profiling component and the Online Access interface of Score-P to send performance metrics for rts's using call-path profiling, as described in Section 6.2.

3.2.2 Periscope Tuning Framework

The Periscope Tuning Framework (PTF) is an extension to Periscope, previously a performance and analysis tool with automatic tuning capabilities. It is a tuning framework that was developed in the Autotune project [8] at the Technische Universität München.

For a tool to effectively and efficiently perform autotuning on modern HPC systems with hundreds of NUMA nodes running highly parallel applications, it should enable fast data aggregation and reduced data analysis time. This is satisfied in PTF using a frontend and a tree-like hierarchical structure of backend components called analysis agents. The framework includes several tuning plugins that can be used to tune individual

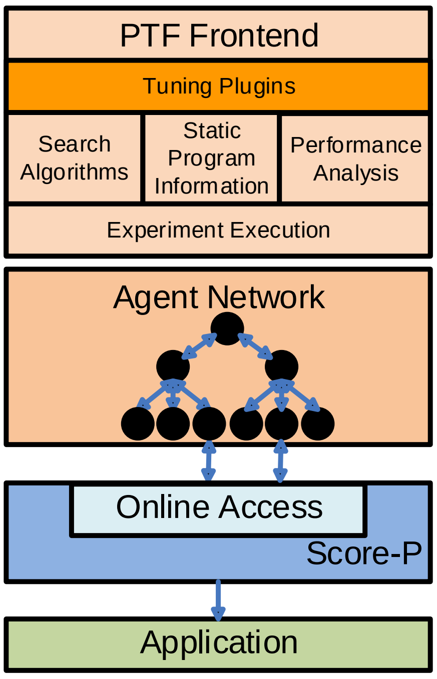


Figure 3.1: Components of PTF and their interaction with Score-P [94].

aspects of HPC applications, such as the selection of optimal compiler flags, or perform parallelism capping for an application. The plugins investigate different configurations of the tuning parameters using search strategies in order to select the best variant. Figure 3.1 [94] depicts a high-level view of the architecture of PTF consisting of the frontend, the tuning plugins, the experiment execution engine and a hierarchy of analysis agents.

PTF performs an iterative online automatic analysis and tuning that requires little to no user intervention. The frontend executes a tuning plugin to tune an application for a given objective, and applies an online search by evaluating experiments with different configurations within a single program run. Each experiment is an individual execution of the phase region in which performance properties are requested from the analysis agents. The PTF analysis agents connect to the application via monitors that can send measurement requests and receive measurement results to and from Score-P via the OA interface. Performance properties include any measurable metric that can be obtained via profiling or a hardware performance measurement tool like PAPI [13], and include execution time, energy consumption, cache misses, branch mispredictions, load imbalances, and MPI-specific profiling data. The performance properties received via the OA interface are finally propagated through the agent hierarchy to the frontend.

Our methodology is based on a two-step approach consisting of DTA and RAT. PTF is used to perform DTA, during which it uses Score-P to request performance measurements. This is discussed in detail in Section 6.2. The current design of PTF only performs static tuning of application code regions, and has no support for dynamic tuning of the rts's. We extend this design using novel plugins to leverage the intra-phase and inter-phase dynamism, as described in Sections 6.2.1 and 6.2.2 respectively.

3.2.3 PAPI

The Performance Application Programming Interface (PAPI) [95] provides tool designers and application engineers a standard API to access hardware performance counters for CPUs, GPUs, chip memory and

I/O independent of the machine and operating system. Hardware counters allow users to examine the relationships between system software performance and hardware events, analyze the bottlenecks in hardware architectures, as well as optimize and scale HPC codes [13]. PAPI is written in C, and provides a standard interface for a number of toolkits that perform profiling, tracing and sampling for HPC applications, such as HPCToolkit, Scalasca, Score-P, TAU, and Vampir [96]. Third-party tools only require the application to be linked to the PAPI library, and handle a single hook to PAPI to access all the hardware performance counters. PAPI also provides fine-grained power management support by providing power and energy measurement counters.

PAPI provides over 100 preset events, which include common CPU predefined events that can be accessed using the simple high-level API, as well as native events comprising all the available events for a specific platform. The fully programmable, low-level API provides a fine-grained measurement and control of both preset and native events from either C or Fortran programs. The high-level API has the advantage of easy use and less setup due to the automatic detection of the components, while the low-level PAPI provides complete control of the entire set of events using event sets.

3.3 Tuning Parameters

To exploit the dynamicity available inside the application and use it to minimize the energy consumption, we distinguish between three different levels of the HPC stack, i.e., hardware parameters, system software parameters, and application-level parameters, and exploit them using the following two approaches:

1. Code-agnostic approach: This includes hardware and system software tuning parameters that are independent of the application, and are described in Sections 3.3.1 and 3.3.2. Thus, no intrinsics of the application code, such as the operations performed or the algorithms used are necessary to exploit these tuning parameters. Simple profiling techniques can be used to detect and define hints to exploit the dynamicity in the application, e.g., defining that different rts's can be identified via different call-paths is sufficient in this approach.

2. Code-aware approach: This includes automatic exploration using the additional knowledge of the application from the developer, specified as domain knowledge specification. Application-level tuning parameters described in Section 3.3.3 allow the developer to provide hints in order to find areas in the application code where dynamicity can be exploited.

The tuning parameters that are relevant to our proposed methodology are summarized in Table 3.1, and are described in more detail in the next sections. It should be emphasized that all of the parameters in Table 3.1 have an impact on the energy-efficiency, and can be tuned at runtime, which is a vital property for dynamic autotuning.

3.3.1 Hardware Tuning Parameters

The most relevant hardware tuning parameters that we exploited are processor-related parameters due to the fact that the processor has the highest power dissipation in the system. We used the processor core frequency and the uncore frequency hardware tuning parameters to perform tuning.

3.3.1.1 Processor core frequency

We perform DVFS as a means to reduce the energy consumption by switching the frequency of the CPU cores. DVFS can be implemented through various means, e.g., soft control by changing the governor, which is available on Linux-based systems as a pluggable infrastructure, and enables the control of CPU frequency settings using defined policies such as performance, powersave, or ondemand, or full control over the frequency settings using the userspace governor to allow arbitrary applications to select the P-state.

Our approach leverages the new feature implemented since the Intel Haswell processor family that P-states can be selected independently for individual cores as opposed to full sockets as was seen in previous processor lines. Moreover, the Haswell processor also introduces a switching window for changing the frequency, where a switching request is given to the processor in a certain time window, at the end of which the frequency is changed. Switching requests that are received after the time window has closed will be executed after the next window is closed, thus causing a delay of up to 500 μs [5].

3.3.1.2 Processor uncore frequency

Haswell allows socket-level granularity to set the uncore frequency using Uncore Frequency Scaling (UFS) to control the transfers to and from memory (communication between processor caches and the DRAM) independently of the CPU core frequencies. The uncore frequency can be switched using the userspace governor by setting machine specific registers (MSRs) using the *x86_adapt* [65] library that allows controlled unprivileged access to MSRs.

3.3.2 System Software Tuning Parameters

The OpenMP standard offers users a means to implement thread-level parallelism through preprocessor statements or pragmas, such as a `parallel for`, which are translated by the compiler into thread-parallel code, and results in the iterations of a loop getting distributed among the threads. We use the API provided by the OpenMP standard to perform DCT to control the behaviour of the OpenMP runtime library by influencing the number of threads used by a parallel region. Typically, each logical core runs a single OpenMP thread, or multiple software threads if it supports simultaneous multi-threading (SMT), and is enforced using the environment variable `OMP_NUM_THREADS`.

By reducing the number of OpenMP threads, we can save the energy consumption, since each hardware thread keeps its own state and can request different C-states, known as thread C-states [30]. Hence, when each thread requests a deeper C-state, the core C-state changes, thus reducing the C-state of unused processor cores from the active C0 state into any of the power-saving C-states C1–C6. Thread and core C-states are identical for cores without SMT support. However, when SMT is enabled, all the threads on a core must request a lower thread C-state to effectively reduce the core C-state.

3.3.3 Application Tuning Parameters

In addition to hardware and system tuning parameters, we exploit Application Tuning Parameters (ATPs) that provide a certain degree of tuning potential using specifications in parts of the code itself that could be used as tuning parameters, e.g., when different implementations of the same algorithm are available, and each has its own impact on performance and energy [97]. ATPs may include the selection between

Table 3.1: Overview of the hardware, system software and application tuning parameters

Level	Tuning Aspect	Tuning Parameter	Granularity
Hardware Parameters	Frequency control	Processor core frequency (DVFS)	Core
		Processor uncore frequency (UFS)	Socket
System Software Parameters	OpenMP threading	Dynamic Concurrency Throttling (DCT)	Process/core
Application Tuning Parameters	Code-paths	Domain decomposition	Application
		Direct and iterative solvers	Application
		Preconditioner	Application

different decomposition algorithms, preconditioners or blocking factors, and are problem-specific. Thus, they can only be handled generically through the definition of variables and their values that are passed to the tuning environment. Since the hardware and software tuning parameters are more interesting for inter-phase dynamism tuning, which is the core of our work, we shall only provide a brief description of the ATPs in Section 6.2.1.2.

3.4 Summary

In this chapter, we presented the formalism used in our proposed approach. We also introduced the core concepts of application dynamism, including intra-phase dynamism arising due to varying characteristics of the significant regions within a single phase, and inter-phase dynamism arising from different execution characteristics between individual phases. We also described how these concepts can be applied to define the tuning potential, which is central to the overall methodology. The result of DTA is the tuning model, which contains groups of rts's with their identifiers, classified into scenarios, and a selector that returns a best configuration for each scenario. By presenting a formal mathematical description of the concepts, we aim to minimize the ambiguity in their definitions.

As a starting point to our methodology, we introduced the Periscope Tuning Framework, a distributed online autotuning framework, which is used to implement the core of our autotuning methodology, and Score-P, a performance measurement infrastructure used by PTF to collect performance metrics. The current design of PTF provides a flexible plugin-based tuning framework, which performs static tuning of program regions. We extended this framework to tune intra-phase and inter-phase application dynamism by leveraging the variance in the characteristics of the application. Moreover, the original design has no concept of rts's, which we introduce for the first time to perform dynamic autotuning. PTF requests performance measurements for the execution time, the objective value and hardware performance counters via the Online Access interface of Score-P. We extended the Score-P OA interface with features to transfer the measurements of call-path based profiling for the rts's to PTF.

We distinguish the tuning parameters between three different levels of the HPC stack, i.e., hardware parameters, system software parameters, and application-level parameters. The parameters that we have identified as being relevant include the processor core and uncore frequencies as hardware tuning parameters, the number of OpenMP threads as a software tuning parameter, and the domain decomposition, preconditioner and types of solvers as the application tuning parameters. The characteristic of all of these tuning parameters is that they can be influenced at runtime.

4 Tuning Methodology

The foundation for our work is the READEX [4] (Runtime Exploitation of Application Dynamism for Energy-efficient Exascale computing) project, which was funded by the European Union Horizon2020 research and innovation programme under grant agreement number 671657. The READEX project was a collaboration between seven partners from academia, HPC resource providers, and the industry, namely Technische Universität Dresden (TUD), Technische Universität München (TUM), Norwegian University of Science and Technology (NTNU), IT4Innovations National Supercomputing Center (IT4I), Irish Centre for High-End Computing (ICHEC), Intel Corporation, and Gesellschaft für numerische Simulation mbH (GNS) with the aim to develop a tools-aided methodology to achieve improvements in the energy-efficiency of dynamic current and future Exascale applications while reducing the effort for tuning.

We first present a general overview of the READEX tool suite in Section 4.1, and describe how it leverages two existing software tools, namely the Periscope Tuning Framework (PTF) and the Score-P infrastructure. We then describe the extensions to the existing PTF framework to leverage application dynamism and perform automatic dynamic tuning at design-time using two novel tuning plugins, namely the *intraphase* and the *interphase* plugins. We also describe in brief the workflow of the READEX Runtime Library (RRL), which is linked to the target application to perform dynamic tuning during application production runs, and how Score-P will serve as a common infrastructure for PTF and RRL by providing instrumentation and measurement capabilities during design-time and runtime tuning.

We follow the high-level description of the workflow with an introduction to the architecture of the READEX methodology, depicting the dependence of different components, including the data-flow between PTF, Score-P and RRL in Section 4.2.

4.1 Overview of the Tuning Workflow

The READEX methodology is based on the system scenario based methodology [98] from the embedded systems domain for energy-efficiency tuning. The system scenario methodology is based on application profiling and code inspection to identify different rts's that have different costs associated with them, e.g., for execution time, or energy consumption. Rts's with similar multi-dimensional system costs, i.e., objective functions are grouped together into scenarios. Optimized configurations are generated for each scenario based on a trade-off between the costs using a Pareto-optimal front. At runtime, the upcoming scenarios are predicted, and configurations are switched for different scenarios if the cost of switching is less than what is saved by switching to the new configuration. Hence, the premise is that the cost of running the application by predicting upcoming scenarios and switching to their best configurations is always less than running the application without switching.

Similarly, the READEX methodology is split into two phases: the Design-Time Analysis (DTA) that is performed during application development, and Runtime Application Tuning (RAT) that is performed during production runs. The end goal is to produce a tuning model as a result of DTA, and use it to perform dynamic switching of the tuning parameters at application runtime to gain energy savings. Thus, READEX brings the system scenario methodology from the embedded world to the HPC domain, and combines the best technologies from opposite ends of the computing spectrum into a tool suite that will contribute towards realizing Exascale performance. Figure 4.1 illustrates the overall workflow of the proposed methodology.

Before performing DTA, the application undergoes some pre-analysis and preparatory steps. In the first step, the HPC application is instrumented using Score-P either by inserting annotations, which are probe functions around different regions that are of interest to tuning using Score-P macros, or by simply letting the compiler perform automatic instrumentation. The only essential manual annotation required is that of the phase region, since this is central to our tuning methodology. Any other annotations are considered domain knowledge that help generate a better tuning model using hints to determine optimal configurations for the rts's. However, the drawback of automatic compiler instrumentation with Score-P is that it causes a high overhead due to the instrumentation of frequently executed fine-granular program regions. To reduce the impact of such fine-granular regions, a tool called *scorep-autofilter* is used to automatically generate a filter file containing the names of the regions that should be omitted from Score-P instrumentation. The tool filters out all the regions in the application whose execution time is less than a certain threshold.

The next step is to automatically identify and characterize the dynamism in the application behaviour. Our tuning methodology only considers applications with significant dynamism since it is based on tuning hardware, system and application-level tuning parameters depending on the dynamism exhibited by different applications regions. A tool called *readex-dyn-detect* is used to analyze the application to determine if there is tuning potential that can be exploited by the methodology. This process is two-fold. First, the tool detects coarse-granular application regions, or significant regions that are worth tuning by selecting only those regions whose exclusive execution time is significant, or above a certain threshold. This is critical because selecting significant regions that are not too fine-granular overcomes the cost of switching. Next, the tool examines the presence of intra-phase and inter-phase application dynamism, and quantifies the tuning potential w.r.t. variations in two metrics: execution time and compute intensity for the instances of the significant regions. The threshold for the amount of variation can be user-specified, as described in Section 6.1.4. If no tuning potential is detected, the tuning process is simply aborted since there is no real benefit of dynamic tuning.

readex-dyn-detect exports the tuning potential summary, the list of significant regions, the intra-phase and inter-phase dynamism information per significant region to a general configuration file in the *xml* format. The configuration file is created from a template, and allows the user to specify the tuning objective, tuning parameters, energy metrics to measure, and the search strategy to use to generate the search space among other information.

After the pre-analysis steps, PTF performs DTA. It reads the configuration file and stores the context information for the significant regions by running a single phase of the target application. It then initializes the *interphase* plugin if the application has inter-phase dynamism, or the *intraphase* plugin in the presence of only intra-phase dynamism.

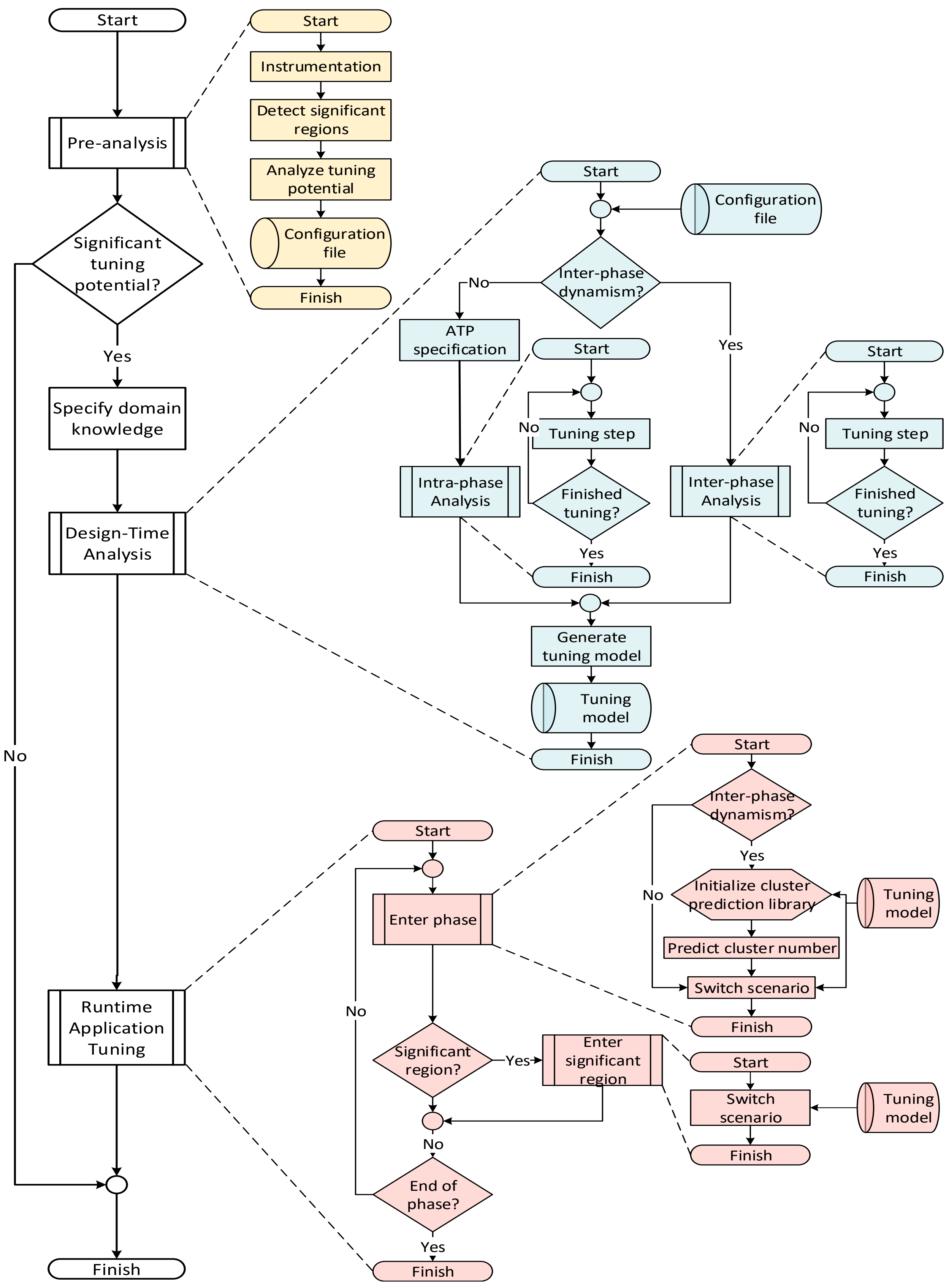


Figure 4.1: A high-level workflow of the autotuning methodology. Pre-analysis steps are colored in orange, Design-Time Analysis (DTA) steps are colored in blue, and the Runtime Application Tuning (RAT) steps are colored in pink.

The tuning plugins run experiments that can evaluate three tuning parameters: CPU frequency, uncore frequency and the number of OpenMP threads within a single program run, where each experiment is a an execution of a single phase. Both plugins first read the ranges (minimum, maximum and the step size) of the tuning parameters, and the objective to tune the application for. The plugins then create a search space to walk the multi-dimensional space of system configurations in one or more tuning steps using a search algorithm to determine which system configuration to evaluate in each experiment. The two plugins however perform different tuning steps, and differ in the way they determine the optimum configurations.

The *interphase* plugin executes the first tuning step with a user-specified number of phases using the default setting of the tuning parameters, and then uses a random search strategy with a uniform probability distribution to generate the search space in the second tuning step. The search strategy randomly picks a configuration in each experiment to see its effect on the application execution, and requests for PAPI hardware performance metrics via Score-P. At the end of the tuning step, it clusters phases that have similar characteristics based on normalized cluster features, namely compute intensity, conditional branch instructions and L2 cache misses, which are derived from the collected PAPI metrics. The clustering is performed by first using a density-based clustering algorithm called DBSCAN, and then a graph-based algorithm called spectral clustering.

In the third tuning step, the plugin uses the random strategy with a probability model based on a Gaussian distribution to select configurations for each phase in a cluster from a search space of configurations that were not executed in the previous tuning step. This step ascertains that a sufficient number of tuning parameters are tested for a cluster of phases before selecting the optimum configurations. Thus, the *interphase* plugin leverages both inter-phase and intra-phase dynamism by determining different cluster-best configurations for clusters of phases, and individual rts-best configurations for rts's within the clusters. Finally, a verification step is performed to ensure that the theoretical computed savings match actual savings incurred after the switching overhead.

The *intraphase* tuning plugin executes the first tuning step using the default settings for the tuning parameters, similar to the *interphase* plugin. It then tunes ATPs using a user-specified search algorithm, e.g., exhaustive or individual search strategy to select the best configuration. The ATPs are implemented via the *ATP library* that provides functions to specify a set of potential values using constraints, which define valid multi-dimensional points, along with a default value. The Omega Calculator [99] is used to solve the system of constraints specified for a given ATP domain. The third tuning step fixes the optimal configuration for the ATPs obtained in the previous tuning step, and then explores the system-level tuning parameters using the selected search strategy. It finally performs a verification step, similar to the *interphase* plugin, and computes the savings.

The best configurations for individual rts's at the end of both plugins, as well as the cluster-best configurations at the end of the *interphase* plugin are then stored in a tuning model. For the *interphase* plugin, additional information about each cluster, such as the phases in that cluster, and the ranges of the features used for clustering are also stored in the tuning model. When the number of rts's that have different best configurations is large, frequently switching between the configurations would result in a corresponding switching overhead w.r.t. both time and energy. Hence, a classifier groups the rts's with similar or identical best found configurations into scenarios using a similarity score that is determined by computing the closeness of system configurations. The tuning model generation was implemented by NTNU, and is described in Section 6.2.3. The tuning model guides the RAT stage to perform dynamic runtime switching of the configurations using the READEX Runtime Library (RRL), implemented by TUD (see Section 6.3).

During production runs, the RRL determines the upcoming scenario based on the current rts by monitoring the application execution, and applies the corresponding optimal configurations for individual rts's and phases using the knowledge in the tuning model. If inter-phase dynamism was detected during application pre-analysis, the best configurations for the unevaluated phases are unknown at runtime, since the *interphase* tuning plugin evaluates only a representative subset of phases during DTA. To avoid setting default system configurations for the unseen phases, the application is linked with the runtime cluster prediction library, which predicts the cluster number of an unseen phase during production runs. To perform runtime cluster prediction, we implemented three cluster predictors: Markov chain predictor, one-bit and two-bit cluster predictors that are based on the respective namesake branch predictors.

RAT also distinguishes seen rts's that were encountered during DTA, and are stored in the tuning model, and unknown or unseen rts's. For unseen rts's, a calibration mechanism is used to find the optimal system configuration based on machine learning algorithms using Q-Learning [16]. When RRL encounters an unknown rts, the algorithm first starts at a certain core and uncore frequency, and then selects a different configuration from the next direct neighbors using a probability. It then measures the energy consumption, and calculates the so-called Q-Value. The algorithm chooses the next configuration based on the Q-Value, and selects it if the cost is smaller than the previous value. We do not discuss the calibration mechanism in more detail, since this work was implemented by TUD, and is not directly related to our extensions or inter-phase tuning.

4.2 Architecture of the Tuning Framework

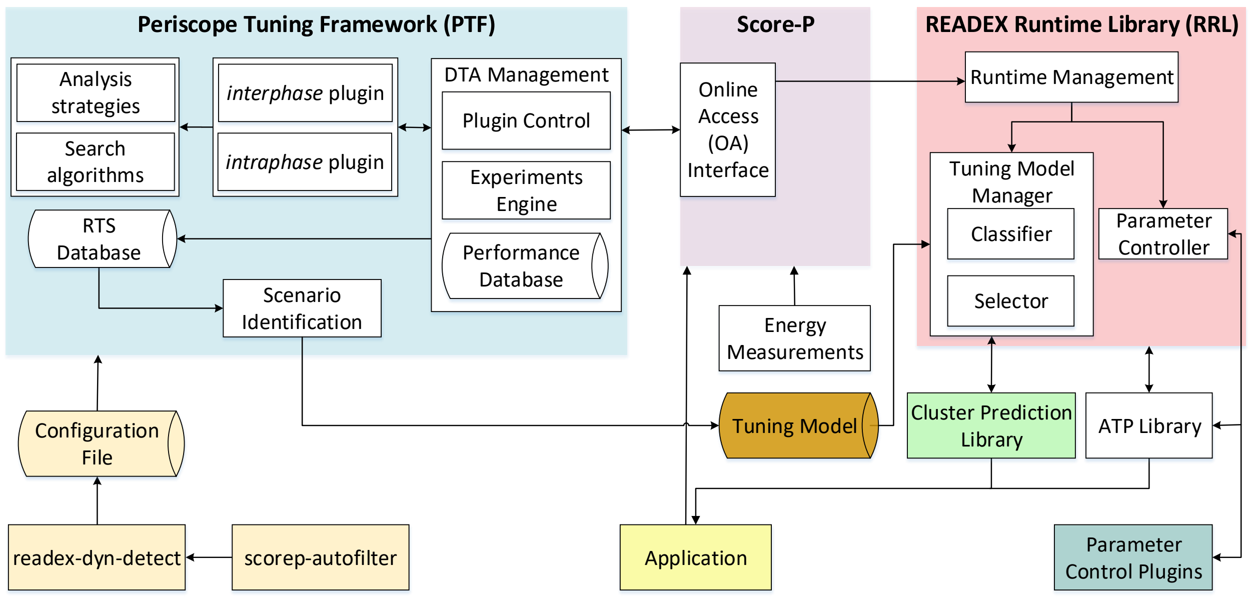


Figure 4.2: The architecture of the READEX methodology, depicting the interaction between PTF, Score-P and RRL. The arrows represent the data flow between the components. Pre-analysis steps are colored in orange, DTA steps are colored in blue, and the RAT steps are colored in pink.

Figure 4.2 illustrates the overall architecture, which depicts the major components that were developed as part of the READEX methodology, and presents the interaction between PTF, Score-P and RRL. The arrows in the figure represent the data flow between the components. The pre-analysis steps consisting of coarse-granular region filtering using *scorep-autofilter* and tuning potential analysis using *readex-dyn-detect* are highlighted in orange to maintain consistency with the high-level workflow in Figure 4.1. The DTA stage performed by PTF is colored in blue, and the RAT stage performed by RRL is colored in pink. We also present the cluster prediction library, which is linked to the application during production runs. To perform ATP tuning for intra-phase analysis, the application is linked with the ATP library, which uses an ATP specification file containing the list of ATPs and the constraints for each parameter.

The DTA Management module controls the overall execution of DTA, and interacts with other PTF components to execute the workflow presented in Figure 4.1. It executes tuning plugins using a state machine to call different stages of the plugin workflow, and incrementally collects information about rts's and invokes methods for storing the data in the RTS database. The tuning plugins determine the best system configuration for a specific tuning aspect, such as the execution time, energy consumption, or compiler flags. The plugins walk the search space using different search strategies, and use analysis strategies to collect performance measurements during each experiment, which are stored in the Performance Database. The Experiments Engine automatically executes individual experiments to evaluate the effect of system configurations on the application execution, and communicates with the RRL via the OA interface to start and stop the application execution. At the end of DTA, the DTA Management module triggers the Scenario Identification module to generate the tuning model by grouping rts's into scenarios and using a selector to pick the best configuration.

To simplify the architecture of RAT, we merged all the components that were not directly extended in our work and pertain solely to the workflow of the RRL at runtime into the Runtime Management module. The Tuning Model Manager (TMM) is used during DTA as well as RAT, and was implemented by TUD and NTNU. At design-time, it saves configurations for individual rts's, and simply returns the configuration that PTF requests. During RAT, the TMM reads the tuning model created during DTA to detect significant regions at application runtime. If inter-phase dynamism was detected during DTA, the cluster prediction library queries the tuning model data from the TMM, and predicts the cluster number. The Runtime Management module then uses the information about an upcoming scenario to send the configuration settings to the Parameter Controller, which switches the tuning parameters via Parameter Control Plugins (PCPs).

4.3 Summary

In this chapter, we introduced the READEX methodology, which forms the foundation of our work. We presented an overview of the entire tuning workflow, consisting of application pre-analysis steps, followed by DTA, tuning model generation, and finally, RAT. We also illustrated the interactions between the major components of our tuning methodology in an architecture diagram.

In our work, we extended the OA interface of Score-P to enable access to the profile measurements for individual rts's from PTF. We created an RTS database in PTF to store the information of the executed rts's received from Score-P via the OA interface. We also extended the DTA Management module to trigger experiments for rts-based tuning. We developed two new plugins to perform inter-phase and intra-phase analysis respectively, and introduced a new analysis strategy to request metrics for inter-phase analysis. We extended the tuning model generation to store the phase and cluster related information for inter-phase tuning, and implemented runtime cluster prediction via the new cluster prediction library.

5

Concepts of Inter-Phase Tuning

Phases play an important role in the tuning methodology as they dramatically reduce the tuning time by enabling the analysis of multiple experiments in a single tuning step without the need for restarting the application. Additionally, phases capture application dynamism, and thus, define the temporal dimension for the inter-phase analysis. Most scientific benchmarks usually execute the same set of instructions by repeatedly calling the same program regions in each iteration. However, complex production-level real-world applications are highly dynamic, and exhibit varying behaviour between different iterations. Executing all the phases with a fixed configuration might result in a suboptimal behaviour and incur an additional expenditure of energy. Tuning for inter-phase dynamism is essential for applications that show variations in the behaviour of individual phases. Therefore, we employ various clustering concepts to target phases that exhibit similar behaviour. This ensures that we can select multiple optimal configurations, i.e., for each cluster of phases as well as individual rts's that are called within the phases of the clusters.

One challenge to this effect is that performing an exhaustive search by walking the search space of the cross-product of all the tuning parameters is simply impractical due to time constraints. Instead, increasing the priority of certain configurations that could potentially result in a higher energy-efficiency enables us to test a higher number of better performing configurations while satisfying the time constraint. Thus, we optimize the search space using the concept of targeted tuning by selecting configurations using a search strategy based on a Gaussian probability distribution. To satisfy the constraint that we do not spend much time in selecting optimal configurations during autotuning, we ensure that only a subset of the entire set of phases is actually run during DTA. These phases are considered to be representative of the application behaviour, and thus enable the selection of near-optimal configurations.

During production runs, the entire application is executed, and a runtime prediction mechanism dynamically predicts the cluster id of all the phases that were not seen during DTA. The RAT stage of the tuning workflow uses the *cluster prediction* library, which provides three kinds of predictors, of which two are based on the one-bit and two-bit dynamic branch prediction schemes, and the other is based on a second-order Markov chain.

In this chapter, we highlight the important concepts of our work with a detailed description of the extensions to the *interphase* tuning plugin. We describe different features that can be used to characterize the similarities between phases, and also present the reasoning behind their selection in Section 5.1. These features are used to perform DBSCAN and spectral clustering. We describe how the *interphase* tuning plugin uses a targeted set of configurations using a Gaussian distribution to select optimal configurations in Section 5.2. We present the extensions to the dynamic tuning component at runtime using a novel cluster prediction library that predicts the characteristics of unseen phases using different cluster predictors in Section 5.3.

5.1 Exploiting the Characteristic Behavior of Phases

The *interphase* tuning plugin selects a representative subset of the progress loop iterations to evaluate during DTA. The plugin executes each phase with a different configuration to evaluate its effect on the behaviour of the phase as well as the tuning objective in order to ultimately select the optimal configuration. The process of selecting the system configurations can be done by using a brute-force strategy or an exhaustive search, which walks the full search space and explores all the combinations of the tuning parameters. While this may result in the optimal system configuration, it is not practical as HPC applications tend to run for a long time, resulting in large tuning costs. Instead, we first use a random search strategy with a uniform probability distribution to select a configuration from a search space of equally likely tuning parameter combinations to test in each experiment. This prevents the search space from exploding.

Each phase is then executed with a randomly selected configuration, and measurements are collected for the objective values as well as hardware performance counters, which are used to characterize the phases and perform clustering. We use PAPI [13] events to collect the hardware counters. Selecting the right metrics is an important first step, since this influences the clustering and ultimately, the selection of optimal configurations. The aim of this step is to use the hardware metrics to group phases with similar characteristics so that a single best configuration can be selected for a cluster.

Selection of metrics

Hardware performance counters help in detecting patterns in an application. For example, AVX instructions expose ALU saturation, or the amount of work done. Thus, codes that have a high compute intensity have the possibility of reusing data in the cache, and hence have fewer Last Level Cache misses (LLC) [100]. Compute intensity can therefore be a good metric to determine the dynamicity in order to perform DVFS. The general formula for the compute intensity of an application is defined as:

$$\texttt{Compute\_Intensity} := \frac{\texttt{[Work\_Done]}}{\texttt{[Bytes\_Transferred]}} \tag{5.1}$$

where, `[Work_Done]` is the total amount of computation performed, and the `[Bytes_Transferred]` metric measures the amount of data moved between the CPU and memory.

A metric that is representative of the work done is the number of retired instructions. However, this is unreliable because it even counts the instructions generated due to the parallelization overhead due to spin-waiting loops that don't do actual work [101]. Hence, it is more effective to count the "useful" instructions that perform actual work by defining `[Work_done]` as the total number of floating-point operations, denoted by FLOPs. Thus, Equation 5.1 may be redefined as the ratio of FLOPS to the LLC misses.

$$\texttt{Compute\_Intensity} := \frac{\texttt{[FLOPs]}}{\texttt{[LLC\_Misses]}} \tag{5.2}$$

Unfortunately, the Intel Haswell architecture does not provide events to measure the FLOPs. However, it allows to count all the AVX instructions including data movement and calculations via the AVX vectorization extension using the `AVX_INSTS.ALL` (Event 0xC6, Umask 0x07) event, which captures all AVX instructions.

Table 5.1: Hardware metrics selected as features for clustering, and their equivalent PAPI/perf events.

Metric	PAPI Event
AVX instructions	`perf_raw::r04C6`
L3 cache misses	`PAPI_L3_TCM`
L2 cache misses	`L2_LINES_IN.ANY`
Conditional branch instructions	`PAPI_BR_CN`

We further refine the measurements to count only the calculations involving floating-point operations using the perf event `perf_raw::r04C6` (Umask 0x04) to represent the actual work done. Thus, we define compute intensity as the ratio of AVX calculation instructions to the L3 cache misses, as shown below:

$$\texttt{Compute\_Intensity} = \frac{\texttt{\#AVX\_INSTS.CALC}}{\texttt{\#L3\_Cache\_Misses}} \tag{5.3}$$

This means that an algorithm that experiences a large number of LLC misses due to a high memory bandwidth results in memory-boundedness where the memory may impede performance.

Compute intensity is however not the best metric to determine the right UFS setting, since the L1 and L2 caches are part of the core domain whereas the L3 cache is part of the uncore domain. In situations when the data or instructions are not present in the L1 and L2 caches, including on-demand and prefetching misses, or when the cache line is invalidated, the CPU loads the cache lines into the L2 cache by reading the data from the main memory. L2 cache misses have a direct effect on the uncore dynamic power consumption, since they cause activity on the ring interconnect and in the L3 cache [102] due to data movement. This means that if all the data were in L2, no performance trade-off is imposed as opposed to an increase in the CPU cycles while reading the data from the main memory. We count the cache lines that are requested and written back to the L2 cache, i.e., the traffic entering the L2 cache, using the PAPI event `L2_LINES_IN.ANY` in order to perform UFS.

The third metric chosen was the number of conditional branch instructions, which represents a change in the control flow within a phase, and indicates if a compute-intensive region is followed by a memory- or IO-intensive region. Conditional branch instructions expose the inherent data-dependency of the application phases, since they require that the conditional code is set and available so that the branch can be executed. They typically incur some pipeline latency, which causes performance penalty due to the stalls in the branch execution. We count the conditional branch instructions using the PAPI event `PAPI_BR_CN`.

Table 5.1 summarizes the features that were selected for characterizing the phases, and their equivalent PAPI/perf events. The listed events are then used to compute the features for clustering, as described in Section 5.1.1. It should be noted that events differ between microarchitectures and across different vendors. Since we do not intend to optimize the tuning framework across all processor architectures, we limit our cluster analysis to the Intel Haswell architecture.

5.1.1 Clustering

Clustering is an unsupervised machine learning technique that groups objects together using their attributes (or features) based on their relationships with other objects, e.g., using pairwise distance or similarity [103]. Unlike classification, which is a supervised learning technique that labels objects with pre-identified categories, clustering does not require assumptions about the categories of objects. Clustering involves the following concepts:

1. Input space: The input space consists of the raw information about the objects, and comprises the associated attributes.
2. Feature space: The feature space consists of the features derived from the attributes of the objects. When there are more dimensions, only the relevant subset of the input attributes are included in the feature space.
3. Similarity space: The similarity space consists of the transformation that translates the features of two objects into a pairwise similarity relationship.
4. Output space: The output space is the representation of the cluster membership of the objects into one of *k* clusters.

The metrics in Table 5.1, i.e., compute intensity, conditional branch instructions and L2 cache misses capture the characteristics of phases, and are used as features for clustering. The mapping of phases into clusters enables the plugin to select a different best configuration for each cluster. A challenge in clustering techniques is finding the right number of clusters for a dataset, since there can be several solutions depending on the granularity of the problem. Hence, we used two clustering algorithms, DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [11] and spectral clustering [12] to cluster phases. First, DBSCAN is performed, since it does not require the number of clusters or make any assumptions about the shape of the clusters a priori, and is robust against noise or measurement outliers. However, it does not consider potential associations or similarities among the outlier points, which spectral clustering models very well.

The plugin first normalizes the features using the *min-max* method. Feature normalization ensures that there is no bias while using features with different data ranges for clustering, and that all features have a similar weight. Without normalization, features with larger values are given more importance, thus degrading the clustering. The *min-max* method is a range normalization method that scales the numeric range of a feature to a [0,1] range, as shown in the following formula:

$$\forall x_i \in X, \quad x_i^{'} = \frac{x_i - min(X)}{max(X) - min(X)}$$

The plugin then uses the normalized features to perform DBSCAN (see Section 5.1.1.1) to group points that are close to each other into high-density regions, and then spectral clustering to determine the associations between unclustered points, as described in Section 5.1.1.2.

5.1.1.1 DBSCAN

DBSCAN is a well-known density-based clustering algorithm that groups points that are close together, resulting in high density regions, and marks points lying in low-density regions as noise. It can detect clusters of arbitrary shapes. The algorithm does not require the number of clusters to be specified beforehand, and needs only two parameters to cluster the data points:

1. **minPts**: *minPts* refers to the minimum number of points that must lie in the neighborhood of a point to define a cluster. The minimum number of points in a cluster is usually defined using $minPts \geq \#features + 1$ [104].
2. **eps**: *eps* is the threshold, or the maximum distance between any two points in order for them to be considered as belonging in the same neighborhood. There are a number of distance metrics, for example, Euclidean distance, Manhattan distance and Minkowski distance that can be used to compute *eps*. This ensures that $\forall p \in D$, the distance between any pair of points is less than or equal to *eps*.

DBSCAN clusters points based on three aspects:

1. The *eps* neighborhood: The *eps* neighborhood of a point p, denoted by $N_{eps}(p)$, is defined as $N_{eps}(p) = \{q \in D | dist(p,q) \leq eps\}$ [11], where D is the set of all data points and *dist(p,q)* is the distance function between points p and q. A point p is defined as a core point if it has at least a minimum of *minPts* number of points within its *eps* neighborhood.

2. Density reachability: Point p is density reachable from q if:

   - $p \in N_{eps}(q)$, meaning p is in the *eps* neighborhood of q.
   - $N_{eps}(q) \geq minPts$, meaning q is a core point (see the definition of core point above).
   - There is a chain of points $p_1, p_2, \ldots, p_n$ where $p_1 = p$ and $p_n = q$ such that p_{i+1} is directly density reachable from p_i.

   A point p is defined as a border point if it is directly density reachable from a core point q and its *eps* neighborhood has fewer than *minPts*, as defined by $|N_{eps}(p)| < minPts$. Finally, a point p is a noise point if it is neither a core point nor a border point, which means that it does not belong to any cluster.

3. Density connectedness; A point p is density connected to a point q if there is a point o such that both p and q are density reachable from o.

Implementation

The algorithm of DBSCAN is presented in pseudocode in Algorithms 1 and 2. For our applications, the parameter *minPts* was chosen to be 4, which means that a cluster has a minimum of four data points. DBSCAN first starts with an arbitrary unvisited point p, and retrieves all points that are density reachable from p. To determine all the points reachable from p, the *eps* neighborhood of p is computed. The estimation of *eps* is not trivial for arbitrary datasets, since this parameter influences the dispersion of a cluster, and hence, must be performed with care.

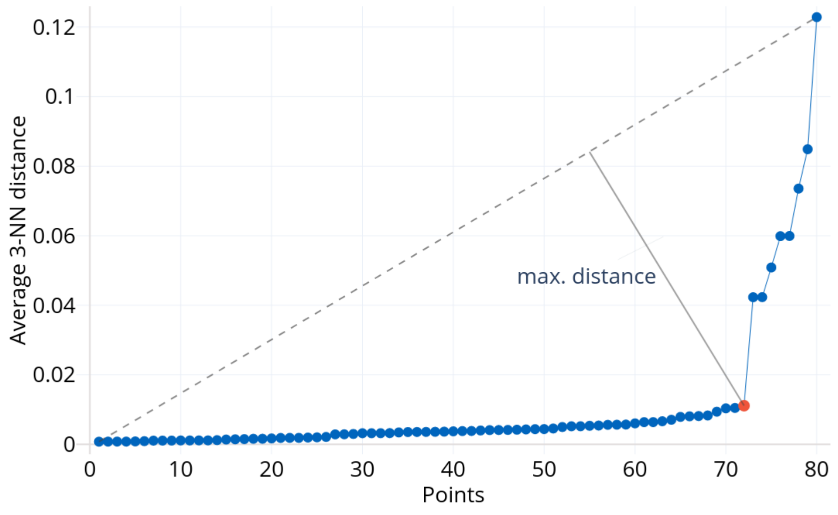


Figure 5.1: Automatic detection of *eps* for DBSCAN. *eps* is the average 3-NN distance of the elbow point colored in red.

In our implementation, *eps* is automatically determined using the *elbow method* [105]. The plugin first computes the average k-NN (k-Nearest Neighbor) distances for all the data points. As we use three-dimensional

Algorithm 1. DBSCAN(*D*, *eps*, *minPts*)

```
Input  : D: Set of all points
         eps: Maximum distance
         minPts: Minimum number of points to form a cluster
Output: A clustering of all points in D
Initialize initial cluster id C = 1
foreach unvisited point $p \in D$ do
    Mark p as Visited
    N = getNeighbors(p, eps), where $dist(p, p') \leq eps$
    if $N \geq minPts$ then
        ExpandCluster(p, N, C, eps, minPts)
        $C \leftarrow C + 1$
    else
        Label p as NOISE
    end
end
```

Algorithm 2. ExpandCluster(*p*, *N*, *C*, *eps*, *minPts*)

```
Input  : p: A visited point in D
         N: Neighbors of p
         C: Cluster id to assign for point p
         eps: Maximum distance
         minPts: Minimum number of points to form a cluster
Add p to cluster C
foreach point $q \in N$ do
    if q is unvisited then
        Mark q as visited
        N' = getNeighbors(q, eps), where $dist(q, q') \leq eps$
        if $N' \geq minPts$ then
            $N = N \bigcup N'$
        end
    end
    if q does not belong to any cluster then
        Add q to cluster C
    end
end
```

data points in our methodology, we use the Euclidean distance as the distance metric because it is easily defined for 3D points. We then compute the average 3-NN distances under the condition that the *minPts* parameter is 4, thus ensuring that point *p* has a minimum of three other points in its neighborhood. The distances are then arranged in the ascending order, and the curve is plotted, as illustrated in Figure 5.1. Since noise points lie further away from the clusters, they have higher 3-NN distances. Thus, the aim is to detect the *elbow*, or a sharp change in the curve of the average 3-NN distances. It occurs at the point that has the maximum distance to the line formed by the points with the minimum and the maximum 3-NN distance, i.e., the first and the last points on the curve. *eps* is computed as the average 3-NN distance of the elbow point [106], which is colored in red in Figure 5.1. It represents a change in density distribution, indicating that only those points whose average 3-NN distances are lower than *eps* will be clustered.

If the current point *p* does not have at least *minPts* in its *eps* neighborhood, it is marked as noise. If *p* is a core point, i.e., it has the specified number of *minPts* in its *eps* neighborhood, a new cluster is started. Each point in its *eps* neighborhood is then marked as visited, and their *eps* neighborhoods are added to *p*'s *eps* neighborhood, as outlined in Algorithm 2. Finally, all the visited points are added into the same cluster. If *p* is a border point, no points are density reachable from *p*, and hence, the next visited point results in a new cluster or becomes a noise point [11]. This process repeats until all the points are marked as visited. After the algorithm successfully terminates, the variable *cluster_num* stores the assigned cluster number for each data point.

5.1.1.2 Spectral Clustering

In the first clustering step, DBSCAN ignores the similarities between the noise points. Although these points don't belong to the clusters obtained as a result of DBSCAN, they could by analyzed to discover associations between them to enhance the tuning result using spectral clustering, which does not make assumptions on the shape of the clusters. It works in the similarity space instead of the original feature space, and relies on the eigen-structure of a similarity matrix derived from pairwise distances to partition similar points into the same cluster [107].

There are several ways of modeling the neighborhood relationships between data points:

- ε-neighborhood graph: In the ε-neighborhood graph [107], all points whose pairwise distances are smaller than ε are connected together. The value of ε should be chosen such that the resulting graph is safely connected. However, this is not easy, since this may result in too large or too small clusters. Moreover, we already use a similar mechanism in DBSCAN to automatically determine the *eps* neighborhood of a point.
- k-Nearest Neighbor graph: In a k-Nearest Neighbor graph, a point *i* is connected to another point *j* if *j* belongs to *i*'s k-Nearest Neighbors.
- Fully connected graph: In a fully connected graph, all points having a similarity above some threshold determined using a similarity measure are connected with each other. We use this method to model the similarities between the noise points.

Spectral clustering requires two inputs:

- The similarity matrix, and
- The number of clusters

The key aspect here is to find a similarity measure that is derived from the cluster features to capture the closeness of a pair of points [103]. A similarity measure $s \in [0,1]$ between two points *i* and *j* is symmetric, i.e., $s(i,j) = s(j,i)$, and has a self-similarity value of 1, i.e., $s(i,i) = 1$, indicating that a point *i* is similar to itself. The goal is to divide the set of points into disjoint subsets with high intra-cluster similarity and low inter-cluster similarity [108].

There are several unnormalized and normalized spectral clustering algorithms available for different types of problems. Unnormalized spectral clustering algorithms use the unnormalized Laplacian matrix, while the normalized spectral clustering algorithms use the normalized graph Laplacian matrix to perform clustering.

We use the well-known Ng, Jordan, Weiss normalized spectral clustering algorithm [12] to cluster the set of points $N = \{1, 2, \ldots, n\}$ into k clusters.

Similarities between individual points can be specified using either a similarity matrix that stores 1 in the $(i, j)^{th}$ entry if points i and j are similar, indicating that there is an edge connecting i and j, or an affinity matrix that specifies that two vertices are connected if the similarity $s(i,j)$ between the corresponding data points is larger than a certain threshold. The similarity matrix is typically used when the relationships between data points are defined using an ε neighborhood or k-NN graph. An affinity matrix is typically used to construct a fully connected graph of a data set, and consists of the set of weights or affinities assigned for the edges between points i and j. It is defined as:

$$A_{ij} = w_{ij}, \quad \forall i, j \in \mathbb{N} \tag{5.4}$$

The weights w_{ij} can be computed in many ways. However, for points that lie in the Euclidean space, the simplest and most commonly used method is the Gaussian similarity function [12]. The original Ng, Jordan, Weiss algorithm used the following similarity function:

$$w_{ij} = e^{(\frac{-d_{ij}^2}{2\sigma^2})} \tag{5.5}$$

where d_{ij} denotes the Euclidean distance between points i and j, and σ is the connectivity or scaling parameter, and controls the width of the neighborhood [107]. σ determines how the affinity A_{ij} falls off with the distance between i and j [12]. The parameter σ may be specified manually using several preset values or automatically.

The Ng, Jordan, Weiss algorithm proposed an automatic way to determine σ by running the algorithm repeatedly for a manually specified range of values, and then selecting the one whose result was the closest to the true clustering of the points. However, this significantly increases the computation time, and poses a big challenge to our autotuning approach of determining best configurations online in as few restarts as possible. Moreover, a single value of σ may not work well for all the data [109]. Therefore, we use the Gaussian similarity function based on "local-scale" [110], which is a self-adaptive parameter, where σ_i is the k^{th} nearest neighbor distance of i, and σ_j is the k^{th} nearest neighborhood distance of j, as shown below:

$$w_{ij} = e^{(\frac{-d_{ij}^2}{\sigma_i \sigma_j})} \tag{5.6}$$

This means that the k-NN distances model the local neighborhood relationships between the data points by setting a threshold on the neighborhood. We selected the value of k as 3, so that σ refers to the 3-NN distance, similar to the DBSCAN algorithm. Ideally, w_{ij} should be 1 or very close to 1 if points i and j are in the same cluster, and close to 0 if the points are in different clusters. Additionally, we set $w_{ii} = 0$ to ensure that there are no self-loops. Computing an affinity matrix that represents the distribution structure is important, since spectral clustering treats the points in the dataset as the vertices of a weighted undirected graph, and the similarity between two vertices as the weights of the edges. Thus, the algorithm essentially converts a clustering problem into a graph partitioning problem, where connected graph components are interpreted as clusters [111], such that the edges within a group have high weights, and the edges between different groups have low weights [107].

Implementation

The spectral clustering algorithm was implemented in `MATLAB R2019a`, and is presented in pseudocode in Algorithms 3 and 4. The algorithm first generates the affinity matrix A using the steps listed in Algorithm 3.

First, it normalizes the features of the noise points in N generated by the DBSCAN algorithm, and computes σ_i and σ_j using the 3-NN Euclidean distances between each pair of points. It then computes the similarity $s(i,j)$ between each pair of points i and j by computing the weight w_{ij} as defined in Equation 5.6. Using the affinity matrix, the algorithm computes the diagonal matrix D containing the sum of each row in its diagonal elements, and constructs the normalized graph Laplacian matrix $L = D^{-1/2}AD^{-1/2}$. It uses the k largest eigenvectors $\lambda_1, \lambda_2, \ldots, \lambda_k$ corresponding to the k largest eigenvalues to construct the matrix X.

Spectral clustering algorithms use K-means as the final step to extract the clusters from the original data points. When K-means is used directly to cluster phases whose natural clustering does not correspond to convex regions, it results in unsatisfactory clustering [12]. Instead, eigenvectors of a Laplacian matrix are used to transform the low-dimensional points into a mapping that can be used by K-means. Typically, for an unnormalized graph Laplacian matrix, the first k eigenvectors would indicate the number of clusters that will be formed, and would hold the information about the k clusters. This is not true for the normalized graph Laplacian matrix used in the Ng, Jordan, Weiss algorithm due to the presence of vertices/nodes that have a low degree, and result in small-valued eigenvectors [107]. Hence, the rows of X are normalized to have unit length to produce matrix Y, thus transforming the low-dimensional data representation into a spectral mapping, or spectral embedding in a k-dimensional space.

Algorithm 3. ComputeAffinityMatrix(N)

```
Input  : $N$: The set of noise points $1, 2, \ldots, n$
Output: $A$: The affinity matrix
1 Normalize the features of the noise points in $N$ generated by the DBSCAN algorithm
2 Compute the $\sigma$ for each point using the 3-NN Euclidean distances
3 Compute the similarity $s(i,j)$ by computing the weight between each pair of points $(i,j)$ using
    Equation 5.6, where the parameter $\sigma$ is the 3-NN distance from the current point
```

Algorithm 4. SpectralClustering(N)

```
Input  : $N$: The set of noise points $1, 2, \ldots, n$
Output: A clustering of all points in $N$
1 Form the affinity matrix $A \in \mathbb{R}^{nxn}$, where $A_{ij}$ is defined in Equation 5.4
2 Compute the diagonal matrix $D$, where $D_{ii}$ is the sum of the elements in $A$'s $i^{th}$ row
3 Construct the normalized graph Laplacian matrix $L = D^{-1/2}AD^{-1/2}$
4 Find the $k$ largest eigenvectors $\lambda_1, \lambda_2, \ldots, \lambda_k$ of $L$, and form the matrix $X = \lambda_1\lambda_2\ldots\lambda_k \in \mathbb{R}^{nxk}$
    containing the eigenvectors as columns
5 Form the matrix $Y$ by normalizing $X$'s rows to have unit length ($Y_{ij} = X_{ij}/(\Sigma_j X_{ij}^2)^{1/2}$)
6 Treat each row of $Y$ as a point in $\mathbb{R}^k$, and cluster them into $k$ clusters via K-means
```

K-means treats each row in Y as a point in the k-dimensional space $\mathbb{R}^k$ [109], and randomly chooses k initial centroids. It repeatedly assigns each point to the cluster with the closest mean, and calculates the new mean for the clusters until convergence, when all the points are finally clustered into k clusters. This way, we overcome the drawback of the simple K-means algorithm that assumes the data distribution to be spherical.

We shall demonstrate the clustering algorithm using a toy example in Figure 5.2. The figure illustrates 150 points that lie in the X-Y plane in the interval [0,1]. The points can be grouped into three natural clusters. First, the affinity matrix is computed using the 3-NN distances of the points, and is illustrated in Figure 5.3.

As we can see, the affinity matrix is block-diagonal, and contains three blocks, indicating that our example

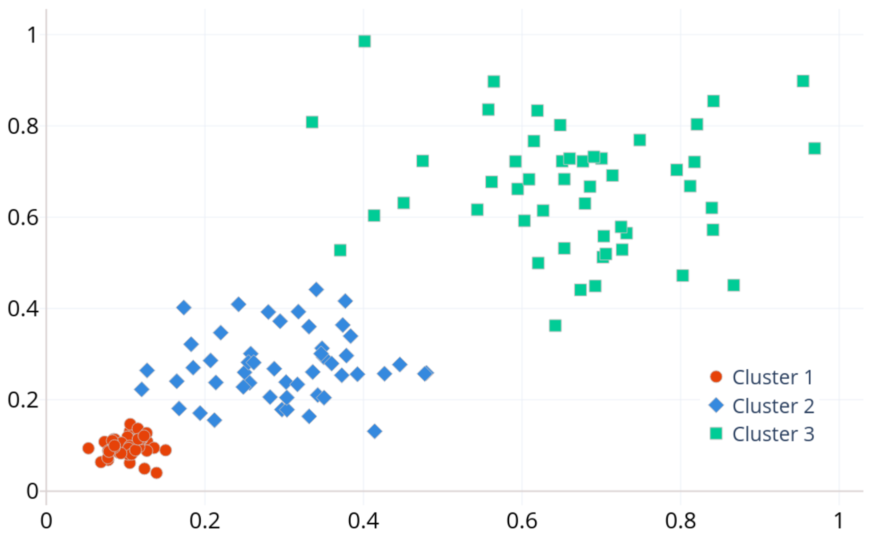


Figure 5.2: Clusters detected for a toy example consisting of 150 points in the [0,1] X-Y plane using spectral clustering.

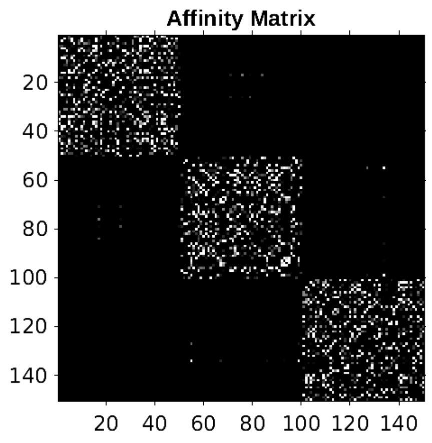


Figure 5.3: Affinity matrix for the toy example computed using the similarity function in Equation 5.6.

has three well-separated clusters, and already gives a general idea of how many clusters can be generated. We then compute the normalized graph Laplacian matrix.

K-means requires the number of clusters k as input for clustering. It should be noted that k here is different from the value of k in the k-NN distance calculation while computing σ in Equation 5.6. The value of k can be provided a priori by the application expert if the application characteristics are already known. Unfortunately, this information is not always available. Thus, we compute k automatically by computing the eigenvalues and eigenvectors of the normalized graph Laplacian, and arranging the eigenvalues in descending order, as illustrated in Figure 5.4. We then use the MATLAB function *findchangepts()* to find points where the mean of the points on the curve changes significantly. These differences or changes in the eigenvalues are called *eigengaps*.

The algorithm uses the concept of matrix perturbation theory, which states that the optimum number of clusters k is the one that maximizes the eigengap, and therefore selects the first point that produces the largest eigengap between the eigenvalues. In Figure 5.5, the points on the graph represent the top 20 eigenvalues for our toy example, where the X-axis represents the number of eigenvalues, and the Y-axis represents the eigenvalues for the normalized graph Laplacian matrix. We see that there are small and insignificant changes between the first three eigenvalues, and a large eigengap between the third and fourth eigenvalues of the normalized graph Laplacian L, as shown by a red line. Thus, k is selected as 3, indicating that K-means clusters the points into three clusters. The column matrix $X = \lambda_1 \lambda_2 \dots \lambda_k$ is then formed from the three largest eigenvectors corresponding to the highest eigenvalues.

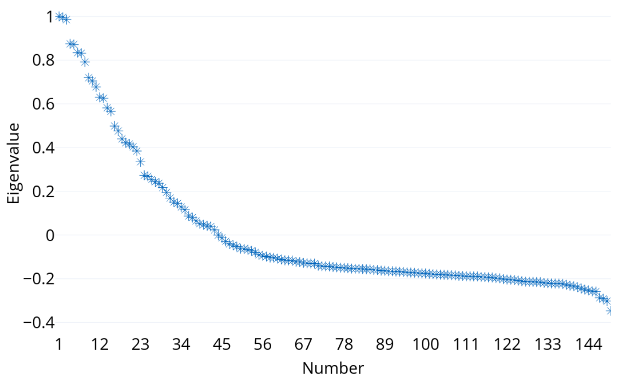


Figure 5.4: Eigenvalues computed for the graph Laplacian matrix for the toy example.

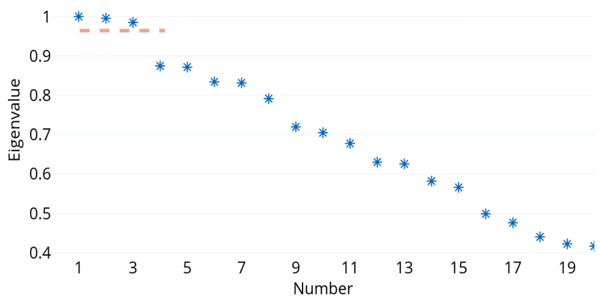


Figure 5.5: A zoomed-in view of the top 20 eigenvalues for the toy example.

Sometimes, DBSCAN may not return a large number of noise points for spectral clustering to be effective. As mentioned earlier, our primary condition for clustering is the presence of at least four points in a cluster in order to proceed to the next steps of the tuning workflow. This is crucial because tuning clusters that have only a few phases will result in a high switching overhead due to switching between every few phases. Therefore, if spectral clustering results in clusters with fewer than four points, the *interphase* plugin simply discards these clusters, and labels the data points as noise.

5.2 Targeted Search

Since we execute only a representative set of the application phases to perform clustering in the previous step, we would have effectively evaluated only a subset of the system configurations. At this point, we can already select near-optimal configurations for the clusters. However, we must do so with relatively low confidence due to the absence of other configurations for comparison.

To overcome this problem, we implemented a targeted tuning step that restarts the application and tests each phase in a cluster with a configuration that has never been executed for any phase in that cluster. A configuration is selected for each phase using a random search strategy based on a normal distribution. This is premised on the idea that certain configurations may be *attractors* or *repellers*. *Attractors* are those configurations that result in a lower normalized energy consumption than a certain threshold, while *repellers* result in higher normalized energy consumption for a phase in a particular cluster. We define the *energy threshold* as the average of the maximum and minimum normalized energy consumption values among all the phases in a particular cluster. It is computed as:

$$\tilde{E} = \frac{E_{max} + E_{min}}{2} \tag{5.7}$$

We redefine attractor configurations as those that result in a normalized energy consumption lower than $\tilde{E}$, and repellers as those resulting in a higher normalized energy consumption than $\tilde{E}$. A configuration may be an attractor for one phase, while being a repeller for another phase. This is due to the fact that a cluster consists of different phases with similar characteristics, such as the computational intensity, but different phase execution times.

For every cluster, the plugin assigns a weight for each configuration of tuning parameters from the cross-product of all the tuning parameters based on its closeness to an attractor or a repeller. A configuration is a set consisting of one combination of the values of the tuning parameters, as described in Section 3.1.2.2, and is represented as $cfg_i = \{tp_1, tp_2, \ldots, tp_n\}$. For the sake of brevity and understanding, we shall represent a configuration simply as x.

For each cluster $c_i \in C$, where $i = 1, 2, \ldots, n$, we define the weight for each configuration x as a function of the influence of the attractors and repellers using:

$$W(x) = \sum_{i \in A} f_{attractor_i} + \sum_{i \in R} f_{repeller_i} \tag{5.8}$$

where A is the set of all attractors and R is the set of all repellers evaluated for the phases of cluster c_i. Functions $f_{attractor}$ and $f_{repeller}$ characterize the influence of an attractive or repelling configuration on the selection of the next combination of tuning parameters. They have the following constraints:

- $f_{attractor}$ and $f_{repeller}$ should be functions of the normalized energy and the distance to an attractor or repeller configuration.
- The curve of the function $f_{attractor}$ for the attractors must gradually increase as we get closer to an attractor configuration, and gradually decrease as we move away from this configuration.
- The curve of the function $f_{repeller}$ for the repellers must strongly decrease as we get closer to a repeller configuration, and gradually increase as we move away from this configuration.

Thus, the influence of the attractors as well as repellers towards a configuration x can be modeled as the sum of individual functions of the difference in their energy consumption and the energy threshold, and the distance between x and an attractor or a repeller x_i. Equation 5.8 can be expanded as:

$$W(x) = \sum_{i \in A} (\tilde{E} - E_i) f_a(dist(x, x_i)) + \sum_{i \in R} (E_i - \tilde{E}) f_r(dist(x, x_i)) \tag{5.9}$$

The first part of the expanded $f_{attractor}$ represents the difference between the energy threshold and the normalized energy for an attractor configuration. For $f_{repeller}$, the first part represents the difference between the normalized energy for a repeller configuration and the energy threshold. The position of E_i is changed for $f_{repeller}$ to keep the difference positive. The second part of $f_{attractor}$ and $f_{repeller}$ is defined by functions f_a and f_r respectively, and are functions of the Euclidean distances between a configuration x and an attractor or repeller configuration x_i. Using f_a and f_r, we want to increase the possibility of picking a configuration around an attractor, and decrease it when the distance between the configuration and the attractor increases. Conversely, we want to decrease the chances of picking a configuration close to a repeller.

The behaviour of f_a and f_r is synonymous with the Gaussian bell curve. A Gaussian function, or a Gaussian, forms a symmetric bell shape, and is of the form $\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$. The parameter $\frac{1}{\sqrt{2\pi\sigma^2}}$ controls the maximum height of the curve, i.e., its peak. μ is the mean, and indicates the position of the center of the peak, and σ is the standard deviation, and controls the width or spread of the bell curve. Smaller values for σ result in sharper or taller peaks, while larger values result in smoother and wider curves. We set the value of σ to 2. If we replace $(x-\mu)^2$ with the squared Euclidean distance $dist(x, x_i)^2$, the resulting function becomes

$\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{dist(x,x_i)^2}{2\sigma^2}}$, which represents the Gaussian as a function of the distance between any configuration x and an attractor or repeller configuration x_i. The Euclidean distance between configurations x and x_i is the same as computing the distance between two points in an n-dimensional space. In our case, n is the number of tuning parameters, which is three. Thus, the distance can be computed as:

$$dist(x,x_i) = \sqrt{(tp_{1,x} - tp_{1,x_i})^2 + (tp_{2,x} - tp_{2,x_i})^2 + (tp_{3,x} - tp_{3,x_i})^2} \tag{5.10}$$

Thus, f_a is defined as:

$$f_a \equiv G(dist(x,x_i)) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{dist(x,x_i)^2}{2\sigma^2}} \tag{5.11}$$

f_r is defined as:

$$f_r \equiv G(0) - G(dist(x,x_i)) \equiv -G(dist(x,x_i)) \tag{5.12}$$

We can observe that the Gaussian in Equation 5.11 is essentially the same Gaussian similarity function that we used in spectral clustering in Equation 5.6. In Equation 5.11, we use the additional parameter $\frac{1}{\sqrt{2\pi\sigma^2}}$ to control the height of the peaks. Thus, a Gaussian is defined for each data point, i.e., an attractor or repeller, and summed up over the set of all attractors A and repellers R to compute the function $W(x)$.

Implementation

Algorithms 5 and 6 list the aforementioned steps to perform targeted tuning of the application. First, the plugin computes the cross-product of the entire search space of tuning parameters. Then, for every cluster identified using DBSCAN and spectral clustering, it computes the energy threshold $\tilde{E}$ using Equation 5.7. It identifies the attractors as the configurations for which the normalized energy is less than $\tilde{E}$, and repellers as the ones whose normalized energy is greater than $\tilde{E}$. The algorithm determines the weight $W(x)$ for every configuration x by computing the sum of $f_{attractor}$ and $f_{repeller}$ for all the attractors and repellers using Equation 5.9.

In order to select the next configuration for a phase in a cluster, we first compute the overall *probability mass function* (pmf) for every possible configuration in the search space using the following formula:

$$P(X = x) = \frac{W(x)}{\sum\limits_{x \in CFG} W(x)} \tag{5.13}$$

It is defined as the probability of selecting a certain configuration x for a phase of a cluster c_i from the set of all configurations CFG. The probabilities are all positive, i.e., $P(x) >= 0$, and sum up to 1. The resulting probabilities for all the configurations are then stored for every phase in the cluster.

Finally, for each phase in the cluster, the plugin randomly selects a configuration to evaluate in an experiment under the following conditions:

1. No phase will be executed with a configuration that was previously executed for any phase in the cluster either in this tuning step or in the previous step.
2. A configuration should be chosen randomly based on the probability mass function.
3. The chosen configuration should be either close to the attractors or far from the repellers.
4. If a previously executed configuration is randomly selected, it is discarded and another configuration is selected.

Algorithm 5. ComputeDiscreteProbability(x, A, R, C)

```
Input  : $x$: A configuration from the cross-product of the tuning parameters
         $A$: The set of all attractors
         $R$: The set of all repellers
         $C$: The set of all clusters
Output: The probability of selecting configuration $x$ for a phase in a cluster
1  Compute the cross-product of the tuning parameters
2  Compute the energy threshold $\tilde{E}$ using the normalized energy consumption using Equation 5.7
3  Identify the attractors and repellers
4  foreach $c_i \in C$ do
5      foreach configuration $x \in CFG$ do
6          ComputeWeightForConfiguration(x, A, R, $\tilde{E}$)
7      end
8      Compute the overall probability mass function
9      foreach configuration $x \in CFG_{c_i}$ do
10         Set the probability $P(X = x)$ to 0, as shown in Equation 5.14
11     end
12     Compute and store the final probability $P(X = x)$ for all the phases of cluster $c_i$
13 end
```

Algorithm 6. ComputeWeightForConfiguration(x, A, R, $\tilde{E}$)

```
Input  : $x$: A configuration from the cross-product of the tuning parameters
         $A$: The set of all attractors
         $R$: The set of all repellers
         $\tilde{E}$: The energy threshold
Output: The weight for configuration $x$
1  foreach $a \in A$ do
2      Evaluate the function $f_{attractor_a}$ using Equations 5.9 and 5.11
3  end
4  foreach $r \in R$ do
5      Evaluate the function $f_{repeller_r}$ using Equations 5.9 and 5.12
6  end
7  Determine the weight $W(x)$ for $x$ by computing the sum of $f_{attractor}$ and $f_{repeller}$ for all the attractors
   and repellers
```

To satisfy the first condition, we set the probability of configuration x to 0 if it was previously evaluated for cluster c_i, as shown below:

$$p(X = x, x \in CFG_{c_i}) = 0 \tag{5.14}$$

By doing so, we ensure that configurations that were already executed for the phases of a cluster will never be picked for evaluation again. Hence, previously executed configurations now become valleys. We then determine the final probabilities for all the configurations in the search space by automatically updating the probabilities for the neighboring configurations to preserve the total sum of individual probabilities as 1.

To satisfy the second and third conditions, we create a hash of each configuration and an integer value associated with it. Then, we use the Mersenne twister random number engine in conjunction with a true random number generator using the C++ `random_device` class to randomly select a hash value. The configuration corresponding to the value is then selected for evaluation. To satisfy the fourth condition, we first select a configuration in each experiment using the random number generator. The plugin then verifies that this configuration has never been selected for any other phase in the current cluster. If the configuration has already been selected, the random number generator repeatedly picks a configuration until it finds one that has never been picked.

We demonstrate the selective tuning step for the toy example presented in Figure 5.2. After performing clustering, we obtain three clusters. Let us assume that the phases in cluster c_i were evaluated with nine unique configurations, out of which five are attractors and four are repellers. The attractors and repellers are listed below in the form $\{$*CPU_freq, uncore_freq*$\}$:

Attractors	**Repellers**
$a_1 = \{1.6, 1.6\}$	$r_1 = \{1.4, 1.7\}$
$a_2 = \{2.1, 1.2\}$	$r_2 = \{1.4, 2.4\}$
$a_3 = \{2.4, 2.3\}$	$r_3 = \{1.8, 3.0\}$
$a_4 = \{2.1, 2.6\}$	$r_4 = \{1.9, 2.9\}$
$a_5 = \{2.1, 2.8\}$	

We can then compute the individual Gaussians for the attractors and repellers using Equations 5.11 and 5.12 respectively, and determine the weight for each configuration in the search space, using Equation 5.9. Figure 5.6 presents a 3D surface-contour plot illustrating the overall weights $W(x)$ in Equation 5.8 by computing f_a over the set of attractors, and f_r over the set of repellers. The X-axis represents the uncore frequency and the Y-axis represents the CPU frequency in GHz. We can see three prominent peaks for the attractors a_2, a_3 and a_4 since they are the strongest attractors, in the sense that they result in the lowest normalized energy consumption. The peak for a_4 is very close to a_3, so we see one big peak in the figure. Similarly, the repellers sit in the valleys, and we can see two prominent valleys, representing r_1 and r_2. Thus, we can say that r_1 and r_2 result in the highest normalized energy consumption.

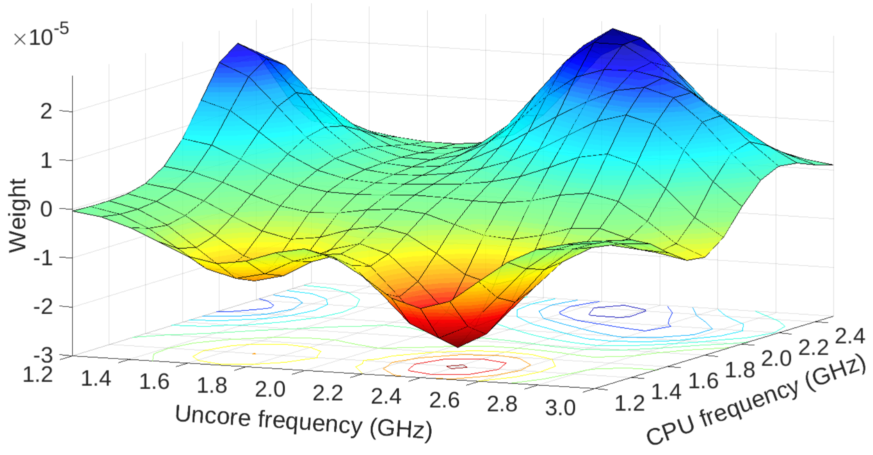


Figure 5.6: The overall weights for the attractors and repellers determined using Equation 5.9.

If we look at the weights for the repellers, they are negative because the function f_r in Equation 5.12 for the repellers results in a curve on the negative Y-axis. Hence, we translate $W(x)$ to the positive Y-axis by shifting the curve up by a constant. The final result for Equation 5.9 is illustrated in Figure 5.7. The attractor configurations lie on the dark blue hills, and the repeller configurations lie in the dark red valleys.

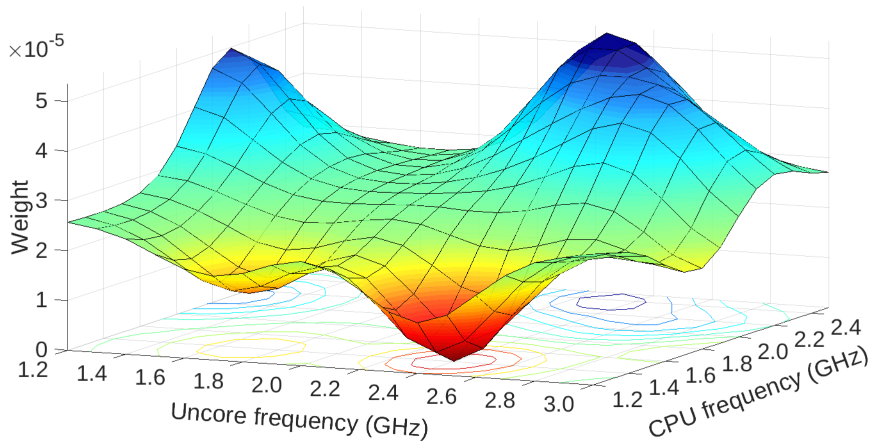


Figure 5.7: The final overall weights for the attractors and repellers obtained after translating the values of f_r to the positive Y-axis. The attractor configurations lie in the blue regions while the repeller configurations lie in the red regions.

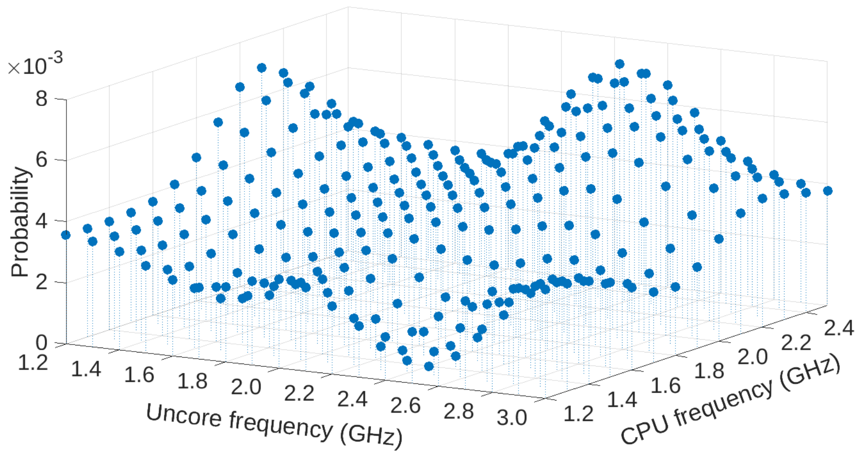


Figure 5.8: The discrete probability mass function obtained for all the configurations of the tuning parameters.

We then compute the discrete pmf for all the phases of the cluster c_i, as illustrated in Figure 5.8. As we can see, the pmf retains the shape of the Gaussian function, and the discrete distribution behaves similarly around an attractor or repeller. Thus, all attractors lie on the peaks and all repellers lie in the valleys of the distribution. The configurations in the neighborhood of the attractors also have a high probability, while the configurations surrounding the repellers have a low probability of being selected. We then set the probabilities for the attractors and repellers to 0 to satisfy the condition that no phase in the cluster will be evaluated with a previously executed configuration, and plot the discrete pmf in Figure 5.9. The figure displays the final probabilities computed for all configurations for a cluster c_i on the Z-axis.

For a better understanding, we also present a 3D surface-contour plot for the final probabilities in Figure 5.10. It should, however, be noted that $P(X = x)$ is a discrete pmf, and hence, the connecting lines in the surface plot do not indicate continuity. The color of the surface varies according to the heights (or probabilities). Attractor configurations have higher probabilities, and are colored in blue. Repeller configu-

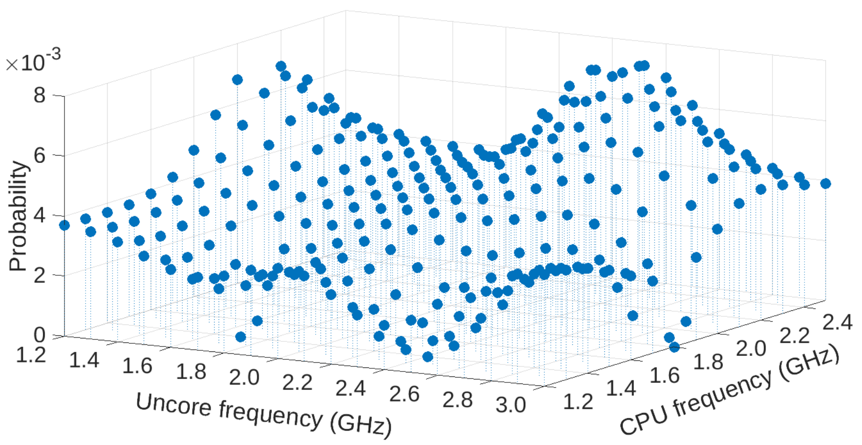


Figure 5.9: The discrete probability mass function obtained under the condition that no phase in the cluster will be evaluated with a previously executed configuration.

rations have much lower probabilities, and are colored in red. If we compare Figures 5.7 and 5.10, we see that in Figure 5.10, the blue surfaces surrounding the attractors have expanded, and the red areas around the repellers have receded to maintain the overall sum of the individual probabilities as 1 in response to setting the probabilities of previously executed configurations to 0.

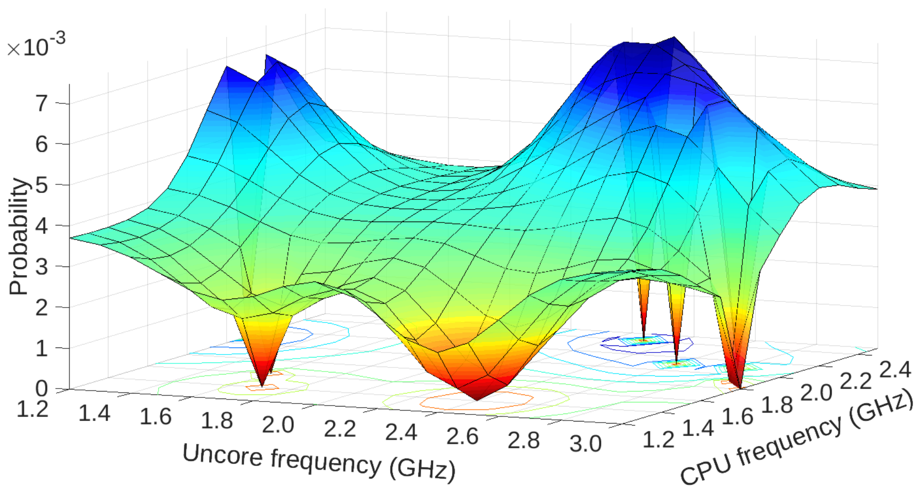


Figure 5.10: The discrete pmf illustrated as a surface-contour plot for better understanding. The lines between the points do not indicate continuity.

During this tuning step, the plugin evaluates the phase using the selected configuration, and requests for the objective value and the hardware performance metrics in each experiment to compute the cluster-best configurations, as well as the rts-specific best configurations for individual rts's of each cluster.

5.3 Dynamic Prediction of Phase Behavior

During DTA, we cluster only a subset of all the application phases, and consider them to be representative of the application behaviour. During production runs, the phases that were already clustered during DTA will be executed with the cluster-best configurations. To apply the appropriate configurations for the remaining phases, we created a library to predict the cluster ids of all the phases that were not executed during DTA. The cluster prediction library is written in C++, and is implemented as a Singleton class. A Fortran or C/C++ application is first linked with the cluster prediction library, and the phase region is annotated with a Score-P phase identifier, as described in Section 6.3. The annotation invokes the library, which calls the *predict_cluster()* method to perform runtime cluster prediction for the current phase.

If the current phase was not evaluated during DTA, i.e., it is an unseen phase, the library predicts the cluster id of the phase using one of the following three prediction mechanisms:

- Markov chain based predictor
- One-bit cluster predictor based on the 1-bit dynamic branch predictor
- Two-bit cluster predictor based on the 2-bit dynamic branch predictor

The one-bit cluster predictor works similar to the one-bit dynamic branch prediction scheme that predicts that the current branch will be taken if it was taken previously. Similarly, the cluster prediction function assigns the current phase to the cluster of the previous phase until the cluster is mispredicted. The two-bit predictor works similar to the two-bit dynamic branch prediction scheme that predicts that the current branch will be taken until it is mispredicted twice in row. Using a similar concept, the cluster prediction function assigns the current phase to the cluster of the previous phase until the cluster is mispredicted twice in a row. The predicted cluster number is returned by the library to the Tuning Model Manager (TMM), which returns the corresponding cluster-best configuration from the tuning model.

Implementation

Algorithm 7 presents the proposed steps performed when the function *predict_cluster()* is called. When the first phase is executed during production runs, the *predict_cluster()* is called for the first time, the cluster prediction library is initialized, and an instance of the cluster prediction class is created. The library requests for the cluster information from the TMM, which reads the tuning model generated at the end of DTA. Then, a number of variables are initialized, namely, *max_dta_phases*, which is set to the number of phases executed during DTA, and *current_phase*, which is a static variable representing the iteration number of the current phase. The PAPI library is then initialized, an event set containing the list of PAPI hardware performance counters from Table 5.1 is created, and measurement collection for the performance counters is started for the phase.

The *predict_cluster()* function then calls the *predict()* method to check if the current phase belongs to the subset of phases executed during DTA using the cluster information returned by the TMM. If the phase was evaluated during DTA, the function simply returns the corresponding cluster number to the RRL. The Runtime Management module requests for the corresponding cluster-best configuration for the current phase, and the configuration is switched dynamically. For the rts's of the significant regions called inside the current phase, the configurations are dynamically switched to the rts-specific best configurations for the current cluster.

Starting from the second phase, the call to *predict_cluster()* method determines whether the cluster prediction library is initialized by checking if the *is_initialized* flag is set. Then, performance counter values for the previous phase are read and aggregated for all the threads and across all the processes, and used

Algorithm 7. *predict_cluster()*

```
Output: The cluster id for the current phase
if !$is\_initialized$ then
    Initialize the cluster prediction library and create an instance
    Request for cluster information from the TMM
    Set $max\_dta\_phases$ to the number of phases executed during DTA
    Set $current\_phase$ to the current phase's iteration number
    Initialize the PAPI library
    Create the event set containing the PAPI hardware performance counters to collect for each phase
    Start the PAPI counters
else
    if $current\_phase \neq 1$ then
        Read PAPI counters for the previous phase
        Aggregate PAPI counters for all threads and processes
        Compute cluster features
    end
    // If current phase is seen during DTA
    if $current\_phase \leq max\_dta\_phases$ then
        Look up the $cluster\_num$ for the current phase from the cluster information
        return $cluster\_num$
    end
    // If current phase is an unseen phase
    if $current\_phase > max\_dta\_phases + 1$ then
        if cluster features $\notin$ ranges of predicted cluster then
            Correct the mispredicted $cluster\_num$ of the previous phase
        else if cluster features $\notin$ ranges of any known cluster then
            Correct the $cluster\_num$ of the previous phase to noise
        end
    end
    if $current\_phase > max\_dta\_phases$ then
        Call the $predict()$ function to predict the cluster id of the current phase using a predictor
          based on either:
            • Second-order Markov chain
            • 1-bit branch prediction scheme, or
            • 2-bit branch prediction scheme
        return $cluster\_num$
    end
end
```

to compute the cluster features for the previous phase. The values of the PAPI counters measured during RAT differ slightly from the values returned by Score-P to PTF during DTA. Hence, the prediction library computes the percentage change and adjusts this difference. The *predict()* method uses these features to perform prediction only if the phase was not seen during DTA. Otherwise, the library simply returns the cluster number of the current phase to the RRL.

For all the unseen phases, an additional step is performed to determine if there was a misprediction of the cluster for the previous phase. The collected PAPI counters are compared against the ranges of the cluster features of the predicted cluster to determine if there was a misprediction. If the values of the features fall within the ranges for the predicted cluster, no action is taken. If the values fall within the ranges of another cluster, the cluster number of the previous phase is corrected. If the values of the cluster features are not in the ranges of any known cluster, the phase is assigned as a noise point.

The following sections describe the three predictors that are used to predict the cluster ids for the unseen phases during production runs. Section 5.3.1 presents the Markov chain predictor, and Section 5.3.2 presents the one-bit and two-bit cluster predictors.

5.3.1 Markov Chain Predictor

Markov chains are one of the well-known and prominent tools that are used to solve complex real-world problems. Markov chains are used in multiple areas, such as predicting the prices of shares, describing weather patterns, predicting the energy consumption, and purchasing patterns [112]. Markov chains are named after the Russian mathematician Andrei Markov, who introduced the concept as a mathematical system defined by the Markov property, which states that given the entire past or history of the occurred events [113], the occurrence of the next event only depends on the current state, but not on the sequence of preceding states [114]. A Markov chain is a stochastic mathematical system, and constitutes a collection of random variables that transition from one state to another according to certain probabilistic rules, while satisfying the Markov property. The term Markov chain is typically used for discrete finite state space, such as discrete time. The discrete time Markov chain has a domain of a discrete set of states. Here, the system is at a certain state at each step, and the state changes randomly between different steps [115], which may be time steps or any other discrete measurements, such as integers or natural numbers. For continuous time, the term Markov process is used. However, many previous works use these terms interchangeably. Since both Markov chains and Markov processes capture the probability of moving to the next state based solely on the current state in the system independently of all the other past states, they are characterized as "memoryless".

A discrete time Markov chain is a sequence of random variables $X_1, X_2, X_3, \ldots, X_n$, where each random variable can take one of the states from the set of all m states constituting the state space $S = x_1, x_2, x_3, \ldots, x_m$. It can be defined using the probabilistic formula:

$$P(X_n = x_n \mid X_1 = x_1, X_2 = x_2, \ldots, X_{n-1} = x_{n-1}) = P(X_n = x_n \mid X_{n-1} = x_{n-1}), \quad x_1, x_2, \ldots, x_n \in S \qquad (5.15)$$

Hence, the probability of predicting the state X_n only depends on the probability of X_{n-1} that precedes it. This satisfies the rule of conditional independence, which means that only the knowledge of the current state is necessary to determine the probability of the next state. Since the future state depends solely on a single previous state, this type of Markov chain is defined as a *first-order* Markov chain.

The conditional probability of the system that is in state x_n at time step t_n can be represented using a *transition matrix* consisting of probabilities derived from transitions, i.e., changes from one state to another. First, a transition frequency matrix is generated by storing the frequency of transitions from a state in row i of the matrix to a state in column j of the matrix, and is represented as N_{ij}. Thus, a random state selected from the matrix has a higher likelihood that it is picked if its transitional frequency is higher relative to the other states. The transition matrix is then generated using:

$$P_{ij} = \frac{N_{ij}}{\sum\limits_{s=1}^{m} N_{is}}, \quad i,j = 1,2,\ldots,m \tag{5.16}$$

where N_{ij} is the number of transitions from state i to state j, and $\sum\limits_{s=1}^{m} N_{is}$ is the total number of transitions from state i in a row of the matrix to all the states in the columns of the matrix. Equation 5.16 essentially converts the transition frequencies into probabilities, as represented below:

$$P = \begin{array}{c c} & \text{Next state} \\ & \begin{array}{cccc} \mathbf{1} & \mathbf{2} & \ldots & \mathbf{m} \end{array} \\ \text{Current state} \begin{array}{c} \mathbf{1} \\ \mathbf{2} \\ \vdots \\ \mathbf{m} \end{array} & \begin{bmatrix} P_{11} & P_{12} & \cdots & P_{1m} \\ P_{21} & P_{22} & \cdots & P_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ P_{m1} & P_{m2} & \cdots & P_{mm} \end{bmatrix} \end{array} \tag{5.17}$$

If the Markov chain has m possible states, the transition matrix will be an mxm square matrix, such that an entry P_{ij} is the probability of transitioning from state i to state j. Additionally, each element of the matrix is non-negative, i.e., $0 \leq P_{ij} \leq 1$. The transition matrix is a stochastic matrix, where the sum of the elements in each row must add up to exactly 1, i.e., $\sum\limits_{j=1}^{m} P_{ij} = 1$, $i = 1,2,\ldots,m$. Thus, each row in the transition matrix represents a probability distribution for state i in a row of the matrix. If we model the system in our previous example in Figure 5.2 using a Markov chain, we obtain 3 possible states, representing the three clusters detected during DTA. Hence, the transition matrix would be a 3x3 matrix.

Sometimes, the next state depends not only on the current state, but also on a previous state, for example when there are patterns while predicting a series. As opposed to first-order Markov chains, where a future state is dependent only on the current state, higher-order Markov chains define that the next state or event depends on a sequence of preceding states. This might be more advantageous in performing predictions as compared to the more restrictive first-order Markov chains. Thus, we can employ more memory by using a second-order Markov model to predict the cluster ids of the current phase during production runs. Second-order Markov chain models can be represented using a lag-1 autoregressive (AR(1)) model in cases where the autocorrelation of the original process is high [116], or via conditional transitional probabilities of different states. In our implementation, we represent the second-order Markov chain using transitional probabilities.

In a second-order Markov chain, the probability of the next state or event depends on the two immediately preceding states. Thus, the probabilities in the transition matrix for a second-order Markov chain require three subscripts, and are defined as:

$$P_{x_{n-2}x_{n-1}x_n} = P(X_n = x_n \mid X_{n-1} = x_{n-1}, X_{n-2} = x_{n-2}), \quad x_1, x_2, \ldots, x_n \in S \tag{5.18}$$

To derive a second-order Markov chain, the transition frequency matrix is first generated by storing the frequency of transitions from the combination of states i and j at time steps $n-2$ and $n-1$ respectively in a row of the matrix to state k in the column of the matrix, which is represented as N_{ijk}. Thus, if we randomly sample from the matrix at time step n, there is a higher likelihood that state k is picked if the value of its transitional frequency is higher relative to the other states. The transition matrix is then generated from the transition frequencies as:

$$P_{ijk} = \frac{N_{ijk}}{\sum_{s=1}^{m} N_{ijs}}, \quad i, j, k = 1, 2, \ldots, m \tag{5.19}$$

where N_{ijk} is the number of transitions to state k given that the system state at time $n-2$ was i, and at $n-1$ was j. $\sum_{s=1}^{m} N_{ijs}$ is the total number of transitions from row ij of the matrix to all the states in the columns of the matrix. The second-order transition matrix is represented below:

$$P = \begin{array}{c} \\ \\ \mathbf{11} \\ \mathbf{12} \\ \mathbf{1m} \\ \mathbf{21} \\ \mathbf{22} \\ \mathbf{2m} \\ \vdots \\ \mathbf{mm} \end{array} \begin{array}{c} \text{Next state} \\ \begin{array}{cccc} \mathbf{1} & \mathbf{2} & \ldots & \mathbf{m} \end{array} \\ \begin{bmatrix} P_{111} & P_{112} & \cdots & P_{11m} \\ P_{121} & P_{122} & \cdots & P_{12m} \\ P_{1m1} & P_{1m2} & \cdots & P_{1mm} \\ P_{211} & P_{212} & \cdots & P_{21m} \\ P_{221} & P_{222} & \cdots & P_{22m} \\ P_{2m1} & P_{2m2} & \cdots & P_{2mm} \\ \vdots & \vdots & \ddots & \vdots \\ P_{mm1} & P_{mm2} & \cdots & P_{mmm} \end{bmatrix} \end{array} \tag{5.20}$$

A state i is called absorbing if it is impossible to leave this state, and is caused when $P_{iii} = 1$ and $P_{iij} = 0$, $\forall j \in m,\ i \neq j$, meaning that all the other elements in the row are 0 [114].

In general, the probabilities for a p-order Markov chain can be estimated as:

$$P(X_n = x_n \mid X_{n-1} = x_{n-1}, X_{n-2} = x_{n-2}, \ldots, X_{n-p} = x_{n-p}), \ \ x_1, x_2, \ldots, x_n \in S \tag{5.21}$$

If we have m states, the transition matrix for a p-order Markov chain is defined by m^pxm transition probabilities from each of the m^p possible preceding states to the m possible subsequent states. Thus, if we have 3 states, a second-order Markov chain would have a transition matrix of the size 9x3, and a total of 27 transitions. It should also be noted that a general problem with higher-order Markov chains is that the number of elements in the matrix rises steeply, thereby limiting their use.

The long-term behavior of a Markov chain can be described using the transition matrix, and an initial probability distribution, represented as a probability vector that describes the initial or starting probability of the states. A probability vector is a row vector of size 1xm^k whose entries are non-negative and sum up to 1 [114]. The row vector of state probabilities at time $t = n + 1$ can be described using the matrix multiplication of the initial probability vector X_n and the transition probability matrix P as shown below:

$$X_{n+1} = X_n P \tag{5.22}$$

For a first-order Markov chain, it can be represented as:

$$X_{n+1} = [\, P_1(n)\ P_2(n)\ \cdots\ P_m(n)\,] \begin{bmatrix} P_{11} & P_{12} & \cdots & P_{1m} \\ P_{21} & P_{22} & \cdots & P_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ P_{m1} & P_{m2} & \cdots & P_{mm} \end{bmatrix}$$

where $P_1(n), P_2(n), \ldots, P_m(n)$ represent the probabilities of states $1, 2 \ldots, m$ at time n.

Summarizing the aforementioned definitions, we define a second-order Markov chain using the following parameters:

- Set of states S : This consists of a finite set of all the values that a state could possibly take. The state space can represent anything from letters, numbers, or weather conditions.
- Set of transitions and the associated transition probabilities: A transition refers to the change of the system from $x_{n-2}x_{n-1}$ to another state x_n at time step n, or staying in the same state between different time steps. The probability associated with this state transition is called transition probability, and is defined by Equation 5.19.
- Initial probability vector: The initial probability vector is a row matrix describing the initial probability distribution for the set of all states of the Markov chain.

The algorithm of the proposed Markov chain predictor is described in the following six steps. To demonstrate the algorithm, we refer to the example in Figure 5.2. Let us assume that only 60% of the total points were selected for clustering using spectral clustering during DTA. To predict the clusters of the remaining points at runtime, we construct a Markov chain to represent the system of known cluster ids using the following steps:

1. **Step 1: Define the states**
   In this step, the algorithm counts the number of clusters returned from the TMM, and computes the number of states in the set S. If some phases were identified as noise during DTA, the total number of states m is set to *num_clusters + 1*, since the Markov chain must model the system inclusive of the noise state. If no noise points were detected during DTA, the number of states is set to *num_clusters*.

   In our toy example, the number of clusters that were detected using spectral clustering is three. Thus, we have a state space of three states, $S = 1, 2, 3$, representing the three cluster ids. If a phase lies in cluster 1, the system is in state 1, and so on. A new point in the X-Y space may belong to any one of the three clusters.

2. **Step 2: Construct the transition frequency matrix**
   In this step, the algorithm constructs the transition frequency matrix of size m^2xm by counting the frequency or the number of transitions from $x_i x_j$ to x_k. To compute this value, the algorithm iterates over the set of phases, and stores the number of transitions from two consecutive cluster ids to the

next cluster id, as shown below:

$$\text{Transition frequency matrix} = \begin{array}{c|c} & \text{Next state} \\ \text{Current state} & \begin{array}{c|ccc} & \mathbf{1} & \mathbf{2} & \mathbf{3} \\ \hline \mathbf{11} & 3 & 1 & 5 \\ \mathbf{12} & 5 & 3 & 2 \\ \mathbf{13} & 5 & 5 & 3 \\ \mathbf{21} & 5 & 4 & 5 \\ \mathbf{22} & 5 & 0 & 1 \\ \mathbf{23} & 1 & 3 & 4 \\ \mathbf{31} & 1 & 4 & 3 \\ \mathbf{32} & 4 & 3 & 5 \\ \mathbf{33} & 3 & 4 & 1 \end{array} \end{array}$$

3. **Step 3: Construct the transition matrix**
   The transition probabilities are computed from the transition frequency matrix using Equation 5.19. We normalize the rows of the transition frequency matrix by dividing each matrix element by the sum of the transition frequencies in its row to obtain the transition matrix, as shown below:

$$\text{Transition probability matrix } P = \begin{array}{c|c} & \text{Next state} \\ \text{Current state} & \begin{array}{c|ccc} & \mathbf{1} & \mathbf{2} & \mathbf{3} \\ \hline \mathbf{11} & 0.3333 & 0.1111 & 0.5556 \\ \mathbf{12} & 0.5000 & 0.3000 & 0.2000 \\ \mathbf{13} & 0.3846 & 0.3846 & 0.2308 \\ \mathbf{21} & 0.3571 & 0.2857 & 0.3571 \\ \mathbf{22} & 0.8333 & 0 & 0.1667 \\ \mathbf{23} & 0.1250 & 0.3750 & 0.5000 \\ \mathbf{31} & 0.1250 & 0.5000 & 0.3750 \\ \mathbf{32} & 0.3333 & 0.2500 & 0.4167 \\ \mathbf{33} & 0.3750 & 0.5000 & 0.1250 \end{array} \end{array}$$

4. **Step 4: Predict the cluster id of the current phase**
   Let us assume that the sequence of the clusters detected during DTA is $1,2,1,3,\ldots,1,3,1$ for our example, where the last two known states are $x_{n-2} = 3$ and $x_{n-1} = 1$. The initial probability vector is $\begin{bmatrix}0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\end{bmatrix}$, since $P_{31} = 1$.

   The probability distribution for the cluster ids is derived by multiplying the initial probability vector with the transition probability matrix, and results in the vector $\begin{bmatrix}0.1250 & 0.5000 & 0.3750\end{bmatrix}$. Thus, the probability that the cluster id for the current phase is *2*, given that the two immediately preceding states are 1 and 3 respectively, is 0.5, and is defined as $P(2|13)$. It is represented by the element P_{312} of the probability matrix.

   We then use the Mersenne twister random number engine in conjunction with a true random number generator using the C++ `random_device` class, similar to Section 6.2.2.3 to randomly pick the next state based on the transition probabilities computed in the previous step. The cluster prediction library returns the predicted cluster number to the RRL, which switches to the corresponding cluster-best configuration for the phase.

The transition matrix above can also be represented in the form of a weighted, directed graph, or a state transition diagram as shown in Figure 5.11. The vertices are the states of the Markov chain, and the edges define the conditional probabilities of going to a state k at time n from the other states. An edge between two vertices exists only if the transition probability P_{ijk} between the vertices is greater than zero. Thus, each edge defines the conditional probability $P(X_n = k|X_{n-1} = j, X_{n-2} = i)$, which is the probability of event k taking place given that j occurred at time step $n-1$ and i occurred at time step $n-2$.

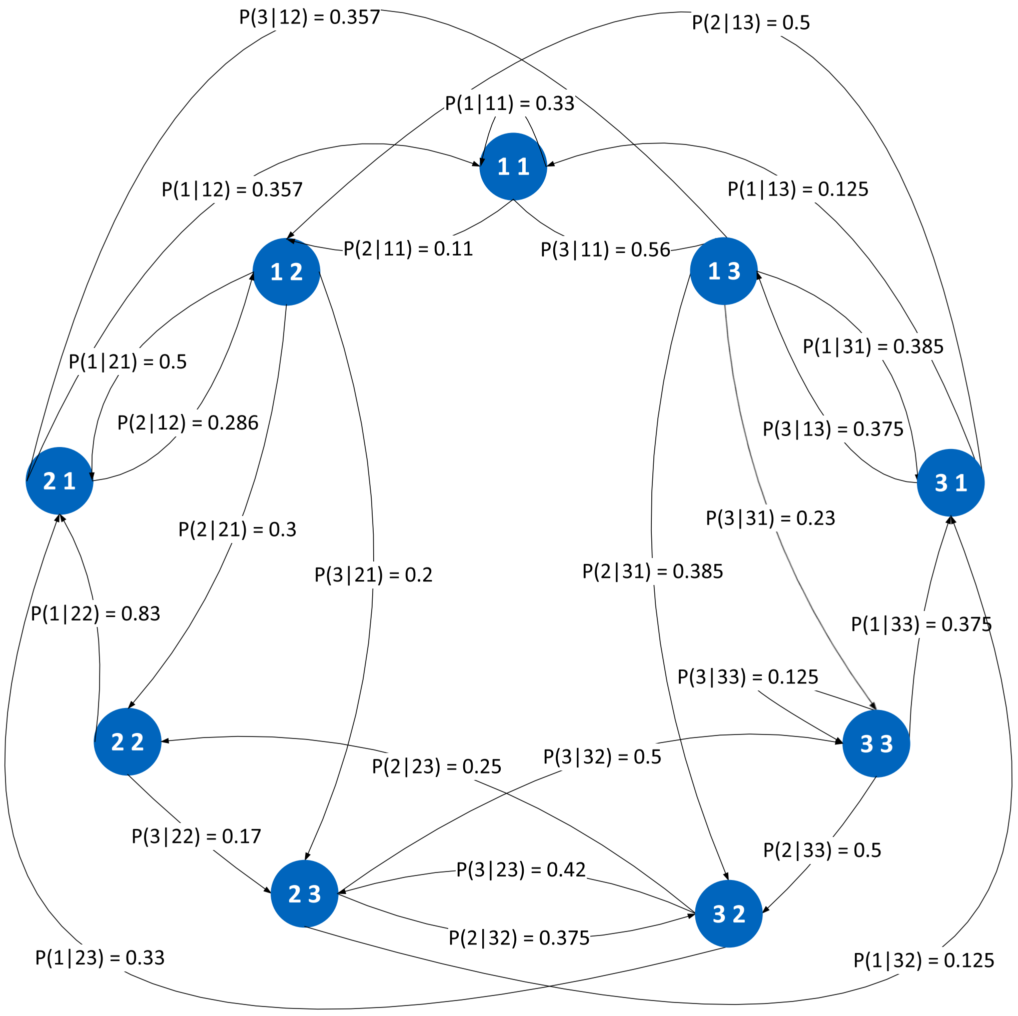


Figure 5.11: State transition diagram of a second-order Markov chain.

To construct the state transition diagram, we first express the Markov chain in the form of a first-order Markov chain [117], such that the state space becomes $S = \{11, 12, 13, 21, 22, 23, 31, 32, 33\}$. An edge from vertex ij to jk represents the conditional probability $P(k|ji)$, defined by the element P_{ijk} in the probability matrix. For example, the probability that the cluster number for the current phase is 2 after the occurrence of 1 at $t = n-1$ and 3 at $t = n-2$ is defined by the edge from 31 to 12, and represented as $P(2|13) = 0.5$.

5. **Step 5: Correct mispredictions**
Initially, the predictor sets the total number states to *num_clusters* if no phase was assigned as noise during DTA. To correct mispredictions, the predictor reassigns the predicted cluster of the previous

phase if its cluster features fall within the ranges of another cluster. If the ranges of the features don't lie within the ranges of any known cluster, the cluster id of the previous phase is marked as noise. If this is the first noise point in the system, a new state is created to represent noise points, and the number of states is updated to $num\_clusters + 1$.

6. **Step 6: Update the transition matrix**
   After correcting the mispredicted cluster of the previous phase, the transition frequency matrix and the resulting transition matrix are updated to reflect the newest edge between two states. The cluster number for the current phase is then randomly selected based on the new transitional probabilities, and returned to RRL, which switches the configuration. Steps 2-6 are then repeated for the remaining phases.

5.3.2 Predictors Based on Dynamic Branch Predictors

Dynamic branch prediction schemes were initially developed to overcome the limitations imposed by control hazards resulting from instruction-level parallelism using pipelined and superscalar instructions. They arise when the CPU cannot determine which instruction to execute next. The one-bit and two-bit cluster predictors are based on the dynamic one-bit and two-bit branch strategies that utilize the execution history of the executed conditional branch instructions in a program. The recent history of branches can be represented using one or two history bits that define whether the conditional branch was taken or not taken, and are stored in the branch history table.

Branch prediction schemes predict the result of a branch instruction using the behaviour of the branches at runtime so that the processor can speculatively fetch (or prefetch) the instruction, and speculatively execute the instruction. Speculative execution can involve eager execution or predictive execution. In eager execution, both paths of the conditional branch are executed. In predictive execution, the path of the conditional branch is predicted, and the next instructions are executed along the predicted path until the actual result is known. If a branch misprediction occurs, the instructions are flushed, and the processor re-fetches the instruction, and the branch history table is updated.

One-bit Predictor

The one-bit predictor is based on the simplest dynamic branch prediction scheme, which stores a single bit in the branch history table, indicating whether the branch was recently taken or not. If the bit is set, i.e., the value is 1, the branch is predicted as taken, and if it is not set, i.e., the value is 0, the branch is predicted as not taken. In the case of a misprediction, the bit state is reversed. Similarly, for each phase that was not evaluated during DTA, the one-bit cluster predictor assigns the current phase to the cluster of the previous phase until it is determined that the previous phase's cluster was mispredicted. The misprediction is confirmed at the beginning of the next phase after comparing the PAPI counters for the previous phase with the ranges of the clusters in the tuning model.

The states of a one-bit predictor are shown in Figure 5.12. A single bit is used to represent two states, `taken` (bit set, i.e., value is 1) and `not taken` (bit not set, i.e., value is 0). This predictor does not work well with loops because it always mispredicts twice, i.e., for the first and the last iterations. For the first iteration, it always predicts `not taken`, and for the last iteration, it always predicts `taken` when in reality, the loop exits.

The overall prediction accuracy of a one-bit branch predictor is not very high. Consider the example in Figure 5.13. We want to predict the sequence `T T T N T T T N T T T N`, shown in the second column using a one-bit counter, with the initial state set to `not taken`, represented as `N`. The predictor returns the

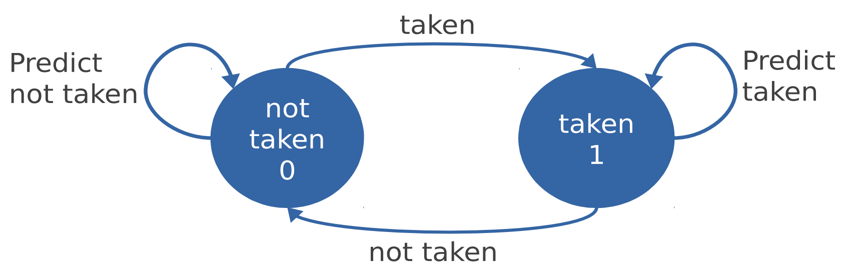


Figure 5.12: State diagram of one-bit dynamic branch prediction.

same state until it is mispredicted, for example, in the first row. It is then updated to the actual outcome shown in the second column. This new value is successively returned until it is mispredicted again. We get six mispredictions, with 50% of the branches being correctly predicted, as shown in the third column.

Initial state: N

Prediction	Outcome	Mispredicted?
N	T	Y
T	T	N
T	T	N
T	N	Y
N	T	Y
T	T	N
T	T	N
T	N	Y
N	T	Y
T	T	N
T	T	N
T	N	Y

Figure 5.13: Example of a one-bit dynamic branch predictor to predict the sequence of branches `T T T N T T T N T T T N`.

Initial sequence: … 2 1 3 1 Initial state: 1

Prediction	Outcome	Mispredicted?
1	2	Y
2	1	Y
1	1	N
1	1	N
1	3	Y
3	1	Y
1	2	Y
2	2	N
2	3	Y
3	2	Y
2	3	Y
3	3	N

Figure 5.14: Analogous example of a one-bit cluster predictor applied to predict the sequence of clusters `2 1 1 1 3 1 2 2 3 2 3 3`.

An analogous example in Figure 5.14 presents the output of the one-bit cluster predictor to predict the sequence of the clusters `2 1 1 1 3 1 2 2 3 2 3 3` for a group of unseen phases during RAT. The initial sequence `...2 1 3 1` refers to the sequence of clusters detected during DTA for the last four phases. Here, the initial state starts with `1`. since it is the cluster id of the last seen phase during DTA. The predictor predicts the same cluster id until it is mispredicted, for example, in the first row. It is then updated to the actual outcome `2`, as shown in the second column. This value is returned until it is mispredicted again. We get eight mispredictions, as shown in the third column, with 33.33% of the clusters being correctly predicted.

Two-bit Predictor

A two-bit dynamic branch prediction scheme uses two bits to represent four states, of which two correspond to `taken` (values `11` and `10`), and the other two correspond to `not taken` (values `00` and `01`). The states of a two-bit predictor are shown in Figure 5.15, where if the initial state is `not taken` with the value `00`, the next prediction would again be `not taken`. If the actual outcome is also `not taken`, it remains in the same state. If the outcome is `taken`, the state transitions into the second `not taken` state with the value `01`. The next prediction would then naturally be `not taken`. If it is the correct prediction, the state transitions back into the `00` state, and if it is a misprediction, the state transitions into the `taken` state, `11`.

The advantage of this approach is that two mispredictions should occur successively before the prediction is corrected. This predictor is more accurate than the one-bit branch predictor, and works well when branches predominantly take one state. Thus, a few atypical branches that result in a temporary change in the direction will not influence the prediction. It does however mispredict thrice for loops because it always mispredicts the first and the second iterations as `not taken`, when they are in fact taken, and the last iteration as `taken` when the loop actually exits.

The two-bit cluster predictor works on a similar technique: it predicts the value of the cluster id of the last phase known to DTA until misprediction occurs twice in a row. The cluster id is then corrected to the right value, which is predicted again for the next phases until two successive mispredictions occur.

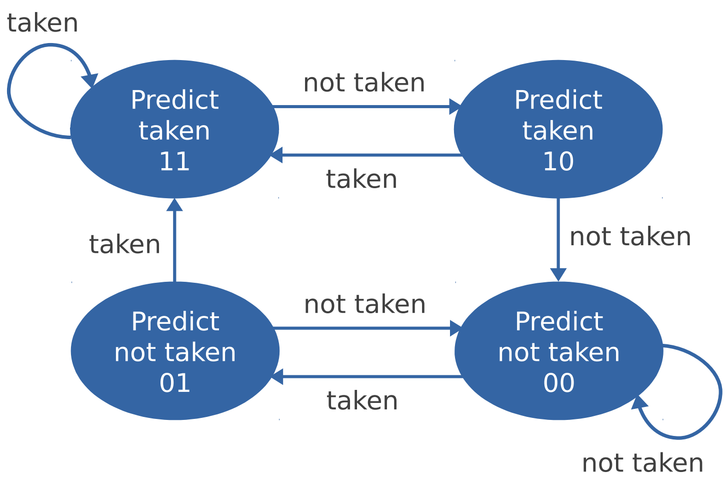


Figure 5.15: State diagram of two-bit dynamic branch prediction.

The overall prediction accuracy of a two-bit branch predictor is higher than the one-bit predictor. Consider the example in Figure 5.16 that shows the predictions and the outcomes for the same sequence `T T T N T T T N T T T N` as before. With the initial state set to `not taken`, represented as `N`, the predictor first returns `not taken` until it is mispredicted twice successively in the second row. It is then corrected to the actual outcome `taken`, as shown in the second column. This new value is predicted until it is successively mispredicted twice. We get a total of five mispredictions, as shown in the third column, and a branch prediction accuracy of 58.3%.

Figure 5.17 presents the output of the two-bit cluster predictor to predict the same sequence of clusters `2 1 1 1 3 1 2 2 3 2 3 3` as before. Here, the initial state starts with `1`, since it is the cluster id of the last phase of DTA, similar to the one-bit technique. The predictor predicts the same value until it is mispredicted twice successively, as shown in the eighth row. It is then updated to the actual outcome `2`. This value is returned until it is again mispredicted twice in a row. We get seven mispredictions, as shown in the third column, with a prediction accuracy of 41.7%.

Initial state: N

Prediction	Outcome	Mispredicted?
N	T	Y
N	T	Y
T	T	N
T	N	Y
T	T	N
T	T	N
T	T	N
T	N	Y
T	T	N
T	T	N
T	T	N
T	N	Y

Figure 5.16: Example of a two-bit dynamic branch predictor to predict the sequence of branches `T T T N T T T N T T T N`.

Initial sequence: ... 2 1 3 1 Initial state: 1

Prediction	Outcome	Mispredicted?
1	2	Y
1	1	N
1	1	N
1	1	N
1	3	Y
1	1	N
1	2	Y
1	2	Y
2	3	Y
2	2	N
2	3	Y
2	3	Y

Figure 5.17: Analogous example of a two-bit cluster predictor applied to predict the sequence of clusters `2 1 1 1 3 1 2 2 3 2 3 3`.

5.4 Summary

In this chapter, we presented the core of our work and the extensions to the READEX tuning methodology for inter-phase tuning. In order to characterize the behaviour of phases for clustering, we presented different metrics, namely AVX calculation instructions, LLC misses, conditional branch instructions, L2 cache misses, and their equivalent PAPI/perf performance counters. We also argued for our reasoning behind the selection of these metrics. Compute intensity exposes the ALU saturation, or the amount of work done, and is calculated as the ratio of AVX calculation instructions to the L3 cache misses. This is useful primarily to determine the DVFS setting. To determine the UFS setting, we use the L2 cache misses, since they represent increased activity on the ring interconnect and the L3 cache, both of which are part of the uncore. The conditional branch instructions represent a change in the control flow, indicating if a compute-intensive region is followed by a memory-intensive region in the program.

We described the main concepts of clustering, and the implementation of DBSCAN and spectral clustering to group similarly behaving phases. DBSCAN requires two inputs, namely *eps* and *minPts*. *eps* is determined automatically using the *elbow* method using the average 3-NN distances between each pair of points. Spectral clustering also requires two inputs, namely the similarity matrix and the number of clusters. The number of clusters is computed automatically using the top k eigenvectors corresponding to the top k eigenvalues. Both algorithms cluster the phases using normalized features computed using the min-max normalization method. The unclustered phases are regarded as noise points.

To evaluate as many configurations as possible, we developed a targeted tuning step that is based on *attractor* and *repeller* configurations, depending on whether they result in an decrease or increase in the normalized energy consumption. A weight is assigned to every configuration in the search space based on the Gaussian distribution, which is a function of the distance of the configuration to an attractor or a repeller. The probability of selecting a configuration increases as we get closer to an attractor, and decreases as we get closer to a repeller. A configuration for each phase in a cluster is selected at random based on the discrete pmf under the condition that the configuration has never been selected before for that cluster.

To save tuning time, only a representative subset of the entire set of application phases are run and clustered during DTA. To predict the cluster number of unseen phases at runtime during RAT, we implemented a cluster prediction library. We developed three cluster predictors, namely one-bit and two-bit predictors inspired by the respective one-bit and two-bit dynamic branch predictors, and a second-order Markov chain based predictor. For each unseen phase, the one-bit predictor predicts the previous cluster number until it is mispredicted, and the two-bit predictor predicts the previous cluster number until it is mispredicted twice successively. The Markov chain predictor computes the transition probability between states x_i at time step $n-1$ and x_j at time step $n-2$ to state x_k at time step n. Then, it predicts the cluster number for the current phase using a random selection based on the transition probability matrix. The cluster predictors return the predicted cluster id to the RRL, which switches the configuration of the phase and the rts's to the corresponding cluster-best and rts-best configurations.

6 Integration into the Tuning Framework

This chapter details the integration of our core work and concepts presented in Chapter 5 for inter-phase tuning into the overall READEX framework.

The tuning methodology comprises the DTA and RAT. As described in Section 4.1, PTF performs DTA, and the RRL performs RAT using Score-P as the common instrumentation and measurement infrastructure. Before DTA starts, a series of preparatory steps are performed by two tools, *scorep-autofilter* and *readex-dyn-detect*. Section 6.1 presents *scorep-autofilter*, which filters fine-granular program regions, followed by Section 6.1.4, which describes the steps performed by *readex-dyn-detect* to identify the significant regions and compute the tuning potential to determine if the tuning effort will potentially result in savings. This chapter also presents the interaction between the *intraphase* and *interphase* tuning plugins and the existing READEX modules. Sections 6.2.1 and 6.2.2 respectively describe the four tuning steps performed by the *intraphase* and the *interphase* plugins at design-time to determine the best configurations for the phase and the rts's.

In addition to the *intraphase* and *interphase* tuning plugins, this chapter describes in Section 6.1.2 the domain knowledge that can be specified by the application expert to enhance the tuning process by annotating special regions or functions to expose application dynamism. Domain knowledge also enables the specification of ATPs, as well as phase, region and input identifiers. The phase and region identifiers distinguish rts's with special characteristics.

At the end of the *intraphase* plugin, a single best configuration for the phase, and best configurations for the rts's are selected. At the end of the *interphase* plugin, a best configuration for each cluster of phases as well as rts-specific best configurations for the rts's of each cluster are determined. Sections 6.2.1.4 and 6.2.2.4 define the static savings for the phase and the rts's and the dynamic savings for the rts's for the *intraphase* and *interphase* tuning plugins respectively.

Section 6.2.3 describes the last step of DTA, i.e., the tuning model generation to store the best configurations for inter-phase and intra-phase tuning. Section 6.3 presents the interactions that take place between the components of the RRL and the cluster prediction library during cluster prediction for an unseen phase.

6.1 Pre-analysis

The pre-analysis step consists of a series of preparatory steps, including application instrumentation, overhead analysis, region filtering, significant region detection, and tuning potential analysis.

6.1.1 Automatic Reduction of Instrumentation Overhead

In the first step, the application is instrumented with Score-P. Instrumentation enables the measurement of performance metrics for various program regions by inserting hooks at the entry and exit points of the regions. Score-P can perform automatic or manual instrumentation for various region types, for example, for all application functions, user-annotated code regions, MPI library calls, and OpenMP parallel regions. For a user that does not have a thorough knowledge of the application code, automatic instrumentation may be preferable with practically no code modification. However, this approach may lead to high overhead due to the instrumentation of frequently executed fine-granular regions, meaning that the execution time per instance is quite low, and thus has an undesirable impact on measurement. Moreover, recording the measurements for such fine-granular regions requires significant space, and analysis takes longer with relatively little to no improvement in quality. For example, in C++ applications, every call to the Standard Template Library (STL) is instrumented, causing a high overhead due to the enter and exit hooks being called hundreds or thousands of times for these functions.

To reduce this overhead, Score-P can be configured with a list of regions that should be omitted from measurement. This, however, does not mean that the regions won't be instrumented, but rather that the region enter and exit hooks will be present, but no measurements will be taken. If users have a good knowledge of the application, they may be able to manually create a filter file containing the regions to omit from being measured. The tool *scorep-autofilter* provides an easier and automatic way to do this, and frees the user from manually determining the regions to insert into the filter file. To use the tool, the application is first instrumented using Score-P, which records measurements for each program region. Although Score-P provides both tracing and profiling features, we only use the profiling option in our methodology. Score-P writes the profile data in the CUBE4 format into a *cubex* file containing a tree of nodes, where each node represents an rts's unique call-path, and profiling data for each call-path, such as the number of visits, execution time or hardware events. *scorep-autofilter* is then invoked by using the command `scorep-autofilter -t <threshold> profile.cubex` to first read the profile stored in *profile.cubex*, and generate a list of frequently executed fine-granular regions.

The granularity of a region is defined as the average execution time of its instances:

$$granularity_{reg} = \frac{t_{incl}^{reg}}{instances\_excl} \tag{6.1}$$

where t_{incl}^{region} is the inclusive execution time for the exclusive instances of region *reg* without counting the instances of nested regions. The tool then filters out all the regions whose granularity is lower than the threshold value in seconds, specified using the `-t` parameter, and adds the names of the functions to the Score-P filter file. The filter file can be specified via the environment variable `SCOREP_FILTERING_FILE` at runtime, or via the `--instrument-filter` flag at compile-time. Score-P filter files have a begin and an end section, followed by the region names that should be excluded from event recording, as shown below:

```
SCOREP_REGION_NAMES_BEGIN
EXCLUDE
add*
binvcrhs*
binvrhs*
copy_x_face*
copy_y_face*
...
SCOREP_REGION_NAMES_END
```

After this step, only the unfiltered regions are candidates for significant region selection in the next step.

6.1.2 Domain Knowledge Specification

The READEX methodology developed the domain knowledge specification interface in order to enhance the tuning process by enabling the application expert to provide user-level specifications to expose domain-level knowledge. The domain knowledge allows the identification of new system scenarios through the information characterizing application dynamism and the additional parameters used for tuning. Domain knowledge may include the following three aspects:

1. Specification of application structure using phase and user region annotations
2. Specification of application characteristics using phase and region identifiers
3. Specification of Application Tuning Parameters (ATPs)

Each of these aspects of the domain knowledge specification is integrated into the DTA workflow at different stages. In addition to the automatic compiler-based instrumentation, instrumentation can be done manually. Manual instrumentation augments automatic instrumentation by enabling the use of region or phase annotations, which can improve the results of tuning by recording additional user-defined metrics. Immediately following the filtering using *scorep-autofilter*, the application expert specifies the application structure, which includes annotating the phase region and additional program regions that should be considered for tuning. For our approach, it is mandatory that the phase region be instrumented. All other annotations are purely to refine the tuning model by differentiating more rts's. The process of specifying the application structure is described in more detail in Section 6.1.3.

The presence of application dynamism is then detected with the help of a tool called *readex-dyn-detect*. The tuning process is either stopped in case of no available dynamism, or the application expert may identify additional application characteristics using region identifiers (see Section 6.1.4), phase identifiers (see Section 6.3), and input identifiers (see Section 6.2.1) to support DTA in generating a more sophisticated tuning model. As mentioned in Section 3.3.3, the tuning methodology also targets ATPs by enabling users to specify parts of the code that can used as tuning parameters, where different implementations of the same algorithm have varying impact on performance and energy. Section 6.2.1.2 describes the ATP specification in more detail.

We shall describe the domain knowledge specification with the help of a single use-case, the MG (MultiGrid) benchmark from the NAS parallel benchmark suite [118], as shown in Listing 6.1. MG uses a V-cycle algorithm to solve a discrete Poisson equation on a 3D grid. After performing an initial residual calculation, each iteration of the time loop of MG executes an entire V-cycle of iterative relaxation and smoothing steps starting from the finest grid, or the highest grid level. First, the result on the current grid level k is projected to the next coarser grid level $k-1$ until the coarsest level is reached, and an approximate solution is computed. The result is then interpolated from the coarser grid levels to the finer grid until the finest grid, where the residual is calculated and a smoother is applied to correct the result. Several V-cycles are executed to solve the equation.

6.1.3 Detection of Significant Regions

After performing the filtering step, the remaining regions are coarse-granular enough to be candidates for tuning. The second step of pre-analysis uses the tool *readex-dyn-detect* to determine the set of significant regions R_{sig} for which DTA will determine best configurations that the RRL will dynamically switch during production runs. Significant regions must be:

1. Coarse enough so that the switching overhead is negligible.
2. Non-nested so that the effect of dynamic tuning is predictable.
3. Coarse-granular enough so as to constitute a major portion of the overall execution time.

Our tuning approach targets applications that have a phase region, or the central progress loop that iteratively performs some computation. Although this loop can be implemented by a standard loop construct of the programming language, automatic detection of the phase region is difficult without analyzing the application source code. Thus, application experts can help provide this information. Score-P offers an *Online Access phase region* annotation to enable external tools like PTF to configure Score-P dynamically when a phase is started. The phase region and user regions are defined using Score-P macros to enclose arbitrary code, and thus can be handled by the tool suite like any other program region.

First, the region handles are defined using the macro `SCOREP_USER_REGION_DEFINE` in line 3 of Listing 6.1. Then, the start and end of the phase region are enclosed in the macros `SCOREP_USER_OA_PHASE_BEGIN` and `SCOREP_USER_OA_PHASE_END` to annotate the body of the iteration loop of MG as the phase region, as shown in lines 7 and 11. The third parameter in the region and phase definitions is "0", which is Score-P specific, and refers to a region without any specific type. The start and end of the phase region warrant a barrier synchronization of all the MPI processes.

After annotating the phase region, the application is run again. The user then invokes *readex-dyn-detect* using the command `readex-dyn-detect -t <granularity_threshold> -p <phase_region_name> profile.cubex`, and specifies a threshold for the minimum granularity of the execution time for a region to be considered significant. The significant region detection algorithm then constructs the transitive closure of the call graph of the application using an adjacency matrix, and selects the leaf node of each candidate region if its execution time is more than the summed up exclusive times of its parents. Otherwise, it selects the parents because they cover more of the execution time, ultimately ending with a list of significant regions.

6.1.4 Analyze Tuning Potential

After detecting the application's significant regions, *readex-dyn-detect* computes the tuning potential by quantifying the application dynamism using the variation in the execution time and compute intensity to detect intra- and inter-phase dynamism. Variation in the execution time between instances of significant regions in an application during its execution is an indication of different resource requirements. The computational intensity models the behaviour of an application based on the load imposed by it on the CPU and

Listing 6.1: Domain knowledge specification for the application structure (phase region and user region), and application characteristics (region identifier) for the MG benchmark.

```
#include "scorep/SCOREP_User.inc"
...
SCOREP_USER_REGION_DEFINE(R1)

do it = 1, max_iter
    ! Phase region begins
    SCOREP_USER_OA_PHASE_BEGIN(R1,"VCycle",0)
    ! Execute a V-Cycle
    call mg3P(...)
    ...
    SCOREP_USER_OA_PHASE_END(R1)
    ! Phase region ends
enddo
...

subroutine mg3P(...)
do k = max_level, min_level+1
    ! Restrict residual from the fine grid to the coarser grid
    call rprj3(...)
enddo

! Compute an approximate solution on the coarsest grid
call psinv(...)
do k = min_level+1, max_level-1
    ! Interpolate the result from coarser to finer grid
    call interpolate(... , k)
    ! Compute the residual for grid level k
    call resid(...)
    ! Apply the smoother to correct the error
    call psinv(...)
enddo
...
end subroutine mg3P

! Interpolate to grid level k
subroutine interpolate(... , k)
SCOREP_USER_REGION_DEFINE(R2)
SCOREP_USER_PARAMETER_DEFINE(intParam)
! User region begins
SCOREP_USER_REGION_BEGIN(R2,"interp",0)
! Region identifier for grid level k
SCOREP_USER_PARAMETER_INT64(intParam,"level",k)
...
SCOREP_USER_REGION_END(R2)
! User region ends
end subroutine interpolate
```

Listing 6.2: Exported environment variables to enable dynamism detection using *readex-dyn-detect*

```
export SCOREP_PROFILING_FORMAT=cube_tuple
export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_L3_TCM
```

the memory, and is calculated using the following formula, which is analogous to the operational intensity used in the roofline model [119].

$$Compute\ Intensity = \frac{Total\ number\ of\ instructions\ executed}{Total\ number\ of\ L3\ cache\ misses} \tag{6.2}$$

The more common definition of compute intensity is based on the floating-point instructions. As these cannot be counted on Intel Haswell, we use the above definition instead. However, it should be noted that this measure of transferred data is very coarse since it does not count all the writes to memory, for example during hardware prefetching.

Computational intensity can directly dictate the effect of the hardware tuning parameters: core frequency and uncore frequency. When the application has a low compute intensity, it may indicate that a memory-intensive region is being executed due to increased L3 cache misses, and would therefore benefit from a higher uncore frequency setting. On the other hand, for a high compute intensity value, it would be desirable to increase the frequency of the CPU cores.

To identify the dynamism, statistical information is requested from Score-P via the cube-tuple profiling format by exporting the environment variable `SCOREP_PROFILING_FORMAT`, as shown in line 1 of Listing 6.2. This extended cube format provides the number of samples, minimum, maximum, average, and deviation of the metrics for the program regions. Additionally, PAPI metrics, as shown in line 2 are requested to derive the compute intensity.

The dynamism detected by *readex-dyn-detect* can be divided into intra-phase dynamism and inter-phase dynamism. The tool analyzes the variation in the time for each significant region by computing the deviation $deviation_r^{reg}$ relative to its mean execution time in percentage in Equation 6.3, and relative to the mean execution time of the phase, defined by $deviation_p^{reg}$ in Equation 6.4. It also computes the variation in the minimum and maximum execution times for the phase region in Equation 6.5.

$$deviation_r^{reg} = \frac{dev\_t_{incl}^{reg}}{mean\_t_{incl}^{reg}} * 100 \tag{6.3}$$

$$deviation_p^{reg} = \frac{dev\_t_{incl}^{reg}}{mean\_t_{incl}^{phase}} * 100 \tag{6.4}$$

$$deviation_p^{phase} = \frac{dev\_t_{incl}^{phase}}{mean\_t_{incl}^{phase}} * 100 \tag{6.5}$$

The tool reports intra-phase dynamism if:

- The region's time variations $deviation_r^{reg}$ and $deviation_p^{reg}$ are larger than a threshold v_t, and the minimal weight of the execution time of a significant region with respect to the phase region according to Equation 6.6 is larger than a threshold v_w.

$$weight = \frac{t_{incl}^{reg}}{t_{incl}^{phase}} * 100 \tag{6.6}$$

- The compute intensity varies across the significant regions, and is larger than a threshold v_i.

The tool reports inter-phase dynamism if:

- The variation $deviation_p^{phase}$ in the minimum and maximum execution time for the phase region is larger than a threshold v_t.

Thus, intra-phase dynamism arises from the variations in the execution time and compute intensity between rts's of significant regions when each rts exhibits different characteristics, resulting in the selection of different optimum configurations. Inter-phase dynamism arises from variations in the execution time between individual phases of the phase region, thus requiring different configurations for individual phases. Listing 6.3 presents an example of the summary of significant regions and the dynamism identified by *readex-dyn-detect* for MG.

The printed output for MG can be divided into three parts. First, lines 3 to 6 list the names of the significant regions detected by *readex-dyn-detect*. Lines 10 to 19 show the profile output for the detected significant regions and the phase region. The columns under the *Significant region information* section present the minimum and maximum execution times, the time deviation in percentage w.r.t. the mean execution time, the absolute value of compute intensity, and the execution time relative to the phase time respectively for each significant region.

The *Phase information* section prints the statistics for the phase region, and shows the minimum, maximum and mean execution time, the deviation in the execution w.r.t. the mean execution time, as well as the variation between the minimum and maximum execution times w.r.t. the mean execution time of the phase. Finally, the tool prints the summary of the dynamism analysis by indicating the presence of inter-phase dynamism due to variation in the execution time of the phases, and the presence of intra-phase dynamism for the significant regions due to the variation in the execution time and compute intensity.

After analyzing the tuning potential, the tool outputs the list of significant regions along with the dynamism information into a configuration file in the *xml* format. The user can then manually annotate the significant regions, which are usually program regions, including subroutines, loops, and structured blocks. Alternatively, the application expert might identify additional regions by combining several calls to different fine-granular functions, or identify certain parts of an algorithm that would otherwise not be a target of tuning because they are not represented by a standard region type, but may have interesting characteristics or tuning potential. Listing 6.1 shows the definition for a user region for the subroutine `interpolate` in lines 40 and 44, where the start and end of the user region are marked with the macros `SCOREP_USER_REGION_BEGIN` and `SCOREP_USER_REGION_END`. The configuration file is then forwarded to PTF, where it is deserialized, and used for DTA.

In addition to the region id and the call-path, which are sent to PTF from Score-P by default, the application expert may specify additional region and phase identifiers to distinguish rts's with special characteristics. These identifiers are specified using Score-P user parameter macros for parameter-based profiling, and can be of type integer or string, which are defined using `SCOREP_USER_PARAMETER_INT64(handle,name,value)` and `SCOREP_USER_PARAMETER_STRING(handle,name,value)` respectively to associate the parameter name with the value.

In Listing 6.1, the size of the grid processed in the call to *interpolate(..., k)* gets bigger when going from the minimum grid level (coarsest) to the maximum (finest). At a certain grid level, the computation switches from being compute-bound to memory-bound. Without region identifiers, the call-path of *interpolate(..., k)* is simply `/VCycle/mg3P/interp`, for which DTA picks one best configuration. We could, however enhance the tuning model by enabling DTA to determine special system configurations for compute- and

Listing 6.3: Summary of the application pre-analysis for MG. Significant intra-phase dynamism due to variation in the execution time and compute intensity was found.

```
...
Significant regions are:
    interp
    psinv
    resid
    rprj3

Significant region information
==============================
Region name   Min(t)   Max(t)   Time   Time Dev.(%Reg)   Ops/L3miss   Weight(%Phase)
rprj3         0.003    0.028    3.433     18.0           171          17
psinv         0.003    0.048    5.263     151.6          145          26
interp        0.002    0.022    2.573     129.7           62          13
resid         0.003    0.054    5.366     152.3           87          26

Phase information
=================
Min         Max         Mean       Time      Dev.(% Phase)     Dyn.(% Phase)
0.283916    0.299874    0.291602   20.4121    1.62227          5.47265

threshold time variation (percent of mean region time): 10.000000
threshold compute intensity deviation (#ops/L3 miss): 10.000000
threshold region importance (percent of phase exec. time): 10.000000

SUMMARY:
========
No inter-phase dynamism

Intra-phase dynamism due to time variation(%) of the following important
significant regions
     rprj3
     psinv
     interp
     resid
Intra-phase dynamism due to variation in the compute intensity of the following
important significant regions
     rprj3
     psinv
     interp
     resid
Writing into the configuration file...
```

memory-bound rts's by specifying a region identifier for the grid level inside the *interpolate* region. First, the identifier is defined in line 38, and then associated with the value of the grid level in line 42. Thus, the region *interpolate* would now have multiple rts's representing different grid levels, for example, `/VCycle/mg3P/interp/level=2`, `/VCycle/mg3P/interp/level=3`,....

We can use a similar strategy to specify phase identifiers. Phase identifiers identify phases with different characteristics or behavior, and can be used in the tuning model to distinguish rts's based on the variation in the phase behavior. Phases that have similar characteristics can be grouped together into a single cluster using phase identifiers, such as the degree of sparsity of the matrix or the arithmetic intensity of the phase region. This enables the prediction or selection of different best configurations for different clusters of phases instead of picking a single static-best configuration for all the phases. Phase identifiers are provided in the same way as region identifiers via Score-P user parameters, but are placed immediately after the `SCOREP_USER_OA_PHASE_BEGIN` macro, and hence are attached to the phase region. Section 6.3 presents phase identifiers in more detail. Phase identifiers have a high impact in selecting the best configurations, and are typically provided by the application expert.

6.2 Design-Time Analysis

The output of *readex-dyn-detect* is stored in a configuration file, which consists of tags through which the user can provide specifications for:

1. Tuning parameters: The *intraphase* and *interphase* tuning plugins support three tuning parameters, namely CPU frequency, uncore frequency and the number of OpenMP threads. The settings for the tuning parameters are specified via the ranges (minimum, maximum, step size, and default value) in MHz for the CPU frequency and uncore frequency, and the lower bound and the step size for OpenMP threads.
2. Objective function: Tuning objectives functions include energy, execution time, CPU energy, EDP, ED2P, and TCO. TCO determines the overall costs of a job as the sum of the energy costs plus the execution time dependent fraction of the HPC system costs, i.e., hardware and software investment as well as maintenance costs and personnel. Each of these objectives also has a normalized version to tune applications that have varying amounts of computation in the phase region, but no varying characteristics, indicating that the changing amounts of work is of the same kind. The objective values are normalized by a metric characterizing the computational load, such as the number of AVX instructions. Multiple objectives functions may also be specified in the configuration file. However, the application is tuned only w.r.t. the first objective, and measurements are collected for the rest.
3. Energy metrics: These include the energy plugin name and the associated metric names. The energy plugin provides energy measurements to Score-P, which transfers them to PTF via the OA interface.
4. Search algorithm: The search algorithm can be either exhaustive, random or individual search strategy. Each of these search algorithms generates the search space differently. The exhaustive search creates the cross-product of all the selected tuning parameters. The advantage is that every combination of the tuning parameters can be explored. However, the search space may explode due to the number of possible configurations. The alternatives are the individual and random search strategies. Individual search explores the tuning parameters individually by assuming independence. First, it finds the best setting for the first tuning parameter, sets this value, and then tunes the next tuning parameter until the best settings for all the parameters are determined. The user can also specify the number of best values of the already evaluated tuning parameter to keep for the next search step using the *keep* tag [8]. The random strategy explores a set number of configurations using the *samples* tag by selecting a

Table 6.1: Identifiers for the significant regions and rts's during DTA.

Name	Description	Identifiers
Regions (user regions and phase region)	The phase region and all significant regions found at design-time	Region id, region name, file name, line number
Runtime situations	Rts's of the significant regions, including user parameters found at design-time	Region name, call-path, user parameter name and value (if user parameters are defined)

configuration randomly from the search space using either a uniform probability distribution or a user-specified distribution. It should be noted that the *interphase* plugin only uses the random strategy to tune the application, and is described in Section 6.2.2.

DTA is then performed by PTF, which consists of a frontend and a hierarchy of analysis agents, in conjunction with Score-P. PTF was extended with two new tuning plugins, i.e., the *intraphase* plugin and the *interphase* plugin to perform intra-phase and inter-phase tuning respectively. The plugins perform DVFS, UFS and DCT to determine the best configurations for the rts's of the significant regions. The *interphase* plugin goes beyond the *intraphase* plugin by leveraging the similarities in the characteristics of phases at application runtime.

To perform DTA, the frontend first executes a single phase and requests the region and rts definitions from Score-P. To enable call-path profiling measurements, we extended the OA interface of Score-P on the monitoring side to transfer profiling data for the rts's. Thus, at every phase enter event, Score-P collects profiling data for the phase and the rts's of the instrumented regions that are called within the phase. At the end of the iteration, i.e., phase exit event, three types of data: flat profile data for regions, call-path profile data for rts's, and the requested metrics are transferred from Score-P to PTF over the OA interface. The flat profile data consists of aggregated measurements and counters for the requested metrics for a significant region whose identifiers are the region name, region id, file in which it is called, and the file line number of the region call site. The call-path data for the rts's consists of the measurements and counters for the requested metrics for individual rts's of the significant regions, and are identified using the calling region name, region id, scorep id, parent scorep id, and user parameter name and value if user parameters are defined. The region id, scorep id and parent scorep ids are the handles that Score-P generates for the regions and the rts's during the application execution. Table 6.1 summarizes the identifiers generated for the significant regions and the rts's during DTA.

After storing the region information, the frontend marks the rts's of significant regions as valid rts's. Only these rts's will then be tuned by the plugins. Partial Calling Context Trees (CCT) [1] are generated at each analysis agent for the MPI processes controlled by it. The partial trees are then gathered in the PTF frontend to create the complete tree that is representative of the application structure as seen by Score-P. We created a new RTS Database to store the rts's received from the OA interface. The phase region represents the root node of the CCT, and its children represent the call sites of the significant regions, i.e., rts's of the significant regions. Each rts is linked to the calling region using the identifiers region id and region name. The user parameter (represented as *parameter_name=value*) defined for a region is represented using a new rts. Thus, in case of the example in Listing 6.1, a separate node representing an rts of *interpolate* is created for each grid level.

[1] A context-sensitive version of a call graph

The frontend then calls the *interphase* plugin if inter-phase dynamism was detected, or the *intraphase* plugin otherwise. Both plugins perform four tuning steps, in which they request measurements and/or perform tuning actions. The following steps describe the general sequence of actions performed by the two tuning plugins.

1. **Initialization**: First, the plugins read the ranges (minimum, maximum and the step size) of the tuning parameters, the search algorithm, and the objective to tune the application for, from the configuration file. If no search strategy is specified, the *intraphase* plugin uses the individual search. The *interphase* plugin only uses the random strategy. If no objective is specified, the default objective function is set to energy, which is the energy consumption of the entire node. The *intraphase* plugin additionally reads the input identifiers from the input specification file as described in Section 6.2.1, and the ATPs from the ATP specification file as described in Section 6.2.1.2.
2. **Experiment creation**: The plugins use the search algorithm to create the search space, and then create individual experiments to test the effects of the system configuration on the application execution, where each experiment is the execution of a single phase. PTF configures an analysis strategy only for the valid rts's, which is executed by the analysis agents after performing the experiments. Every analysis strategy processes different measurements. For example, the *interphase* plugin requests for the `InterphaseProps` strategy that processes both PAPI hardware metrics and the requested objectives.
3. **Experiment execution**: In each experiment, PTF sends two types of requests to Score-P via the OA interface:

    - Tuning request: PTF generates tuning requests for each rts and the phase to configure the tuning parameters in order to measure the effect of the new configuration on the tuning objective. The tuning request is read by the OA interface in Score-P, and sent to the RRL, which applies the new configuration.

      The rts tuning request consists of four inputs: the region id where the new configuration should be applied, the list of additional user parameters, the list of tuning parameters and their values, and the ranks for which the configuration should be applied. Listing 6.4 shows an rts tuning request sent from PTF for the *interpolate* region with a user-defined parameter for the grid level for the MG code example in Listing 6.1. The request encodes a switch of the current core frequency to 2.0 GHz, the uncore frequency to 2.5 GHz, and the number of threads to 1 for the rts whose call-path is */VCycle/mg3P/interp/level=3*. The route to this rts is determined by the rts call stack, which consists of four elements, starting from the phase region *VCycle*, whose Score-P region id is 39, followed by *mg3P*, whose region id is 30, *interp* with region id 35, and finally the user parameter with region id 36.

Listing 6.4: Rts tuning request sent from PTF to Score-P.

```
RTSTUNINGREQUESTS((39,INTPARAMS=(),UINTPARAMS=(),STRINGPARAMS=()),(30,INTPARAMS=⌋
↪  (),UINTPARAMS=(),STRINGPARAMS=()),(35,INTPARAMS=(),UINTPARAMS=(),STRINGPARAMS=()),⌋
↪  (36,INTPARAMS=(”level”=3),UINTPARAMS=(),STRINGPARAMS=()))=⌋
↪  ("NUMTHREADS"=1,"CPU_FREQ"=2000,"UNCORE_FREQ"=2500)
```

  - Measurement requests: The plugin sends measurement requests to the analysis agents for several global metrics, such as execution time, PAPI counters or energy consumption. Listing 6.5 shows an example of a measurement request sent from PTF to Score-P through the OA interface. The request triggers Score-P to collect node energy, defined as `x86_energy/BLADE/E` from the *x86_energy_sync_plugin* [120]:

Listing 6.5: Measurement request sent from PTF to Score-P.

```
REQUEST[0] GLOBAL METRIC PLUGIN "x86_energy_sync_plugin" "x86_energy/BLADE/E"
```

The OA interface of Score-P parses the metric request, and returns the measurements to PTF after executing the tuning request. Each analysis agent stores the measurements returned by Score-P for those MPI processes controlled by it. The measurements for the phase and the objective values for the rts's are then stored in the Performance Database, and accessed in the form of properties by the analysis strategies to evaluate the objective functions. The objective value is then propagated to the tuning plugin. The Performance Database also supports generic energy counters for core and node energy to enable easier porting to other systems.

If energy is selected as the objective, Score-P returns the consumed energy via a designated single process on each node, since energy is a global metric. The plugins then compute the node energy consumption by aggregating the values returned by the designated processes across all the nodes for an MPI application.

4. **Process results**: When there are no more experiments left to be executed, the plugins determine the best setting of the tuning parameters for the selected objective, and compute the savings incurred as a result of DTA.

Both plugins perform steps 1-4 in the first tuning step. At the end of a tuning step, PTF restarts the application, and repeats steps 2-4 during each subsequent tuning step. Sections 6.2.1 and 6.2.2 describe each tuning step performed by the *intraphase* plugin and *interphase* plugin respectively in more detail.

6.2.1 Tuning Intra-Phase Dynamism

PTF tunes the intra-phase dynamism by executing the *intraphase* plugin when there are no changes in the dynamic characteristics across the sequence of phases. The *intraphase* plugin executes four tuning steps: default execution, ATP tuning, system parameters tuning, and verification, as illustrated in Figure 6.1. First, the plugin executes a single phase of the application using the default settings of the tuning parameters. Next, it determines the optimal configuration of the ATPs, and then the optimal configurations of the system-level tuning parameters for the phase and the rts's. Finally, the plugin checks the variations in the tuning results, and computes the energy savings. The tuning steps are described in detail in Sections 6.2.1.1- 6.2.1.4.

The plugin implements novel features, such as tuning model merging for different input sets, and tuning of ATPs. The plugin not only tunes the application for a single input, but also learns from running the application for different inputs. We characterize variations in application executions for different input sets via domain knowledge specification for input identifiers, which leads to the identification of more rts's with different characteristics. Input identifiers may be simple, such as the grid size of the application domain and the number of processors, or complex, like the number of contact points in metal forming simulations like INDEED. The domain knowledge specification interface allows the application expert to provide input identifiers in an accompanying input specification file in the form of key-value pairs, as shown in Listing 6.6.

The input identifier specification files are read by both PTF during DTA and RRL during runtime tuning. PTF reads the specification file called `IID SPEC` via the command line option using the flag `--input-desc="IID SPEC"`, and RRL interprets them via an environment variable, as described in Section 6.3.

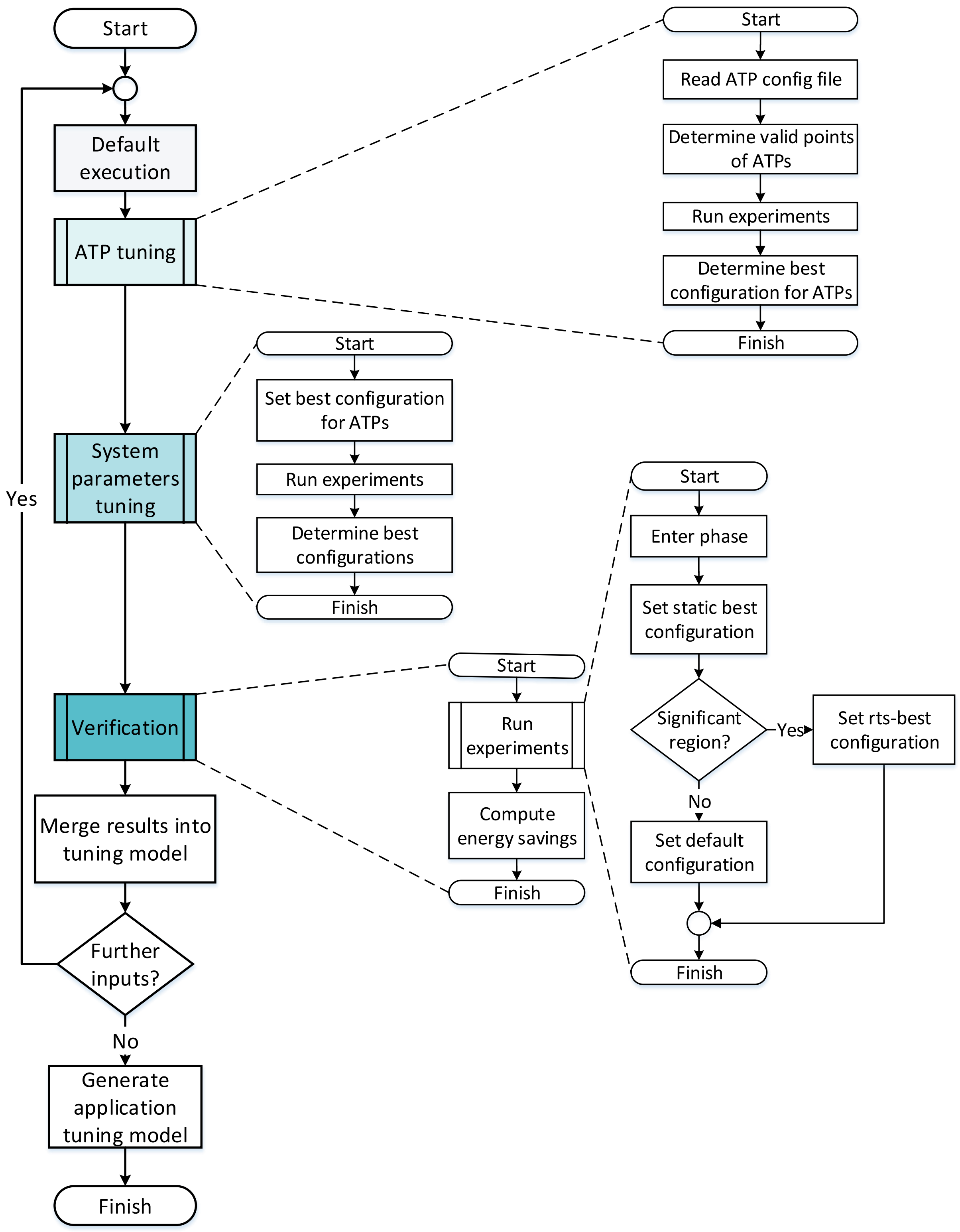

Start
Default execution
ATP tuning
System parameters tuning
Verification
Merge results into tuning model
Further inputs?
Yes
No
Generate application tuning model
Finish
Start
Read ATP config file
Determine valid points of ATPs
Run experiments
Determine best configuration for ATPs
Finish
Start
Set best configuration for ATPs
Run experiments
Determine best configurations
Finish
Start
Run experiments
Compute energy savings
Finish
Start
Enter phase
Set static best configuration
Significant region?
Yes
Set rts-best configuration
No
Set default configuration
Finish

Figure 6.1: The workflow of the *intraphase* tuning plugin, consisting of four tuning steps: default execution, ATP tuning, system parameters tuning, and verification. The plugin merges the tuning models for new inputs, and generates the final application tuning model at the end of DTA.

Listing 6.6: Input identifier specification file IID SPEC.

identifier name	:	***<value>***
ContactPoints	:	<simple, complex>
ProblemSize	:	<256 256 256>

It is not necessary to pass the same input identifier specification file to both PTF and RRL. If input identifiers from one input file are missing in an other input file, the missing identifiers are handled in DTA with a default value. In the MG benchmark in Listing 6.1, when the grid level switches from a coarser grid to a finer grid, the compute intensity characteristics might change from compute-bound to memory-bound. The grid level at which this switch occurs depends, for example, on the grid (or problem) size given in the input file. Thus, the user parameter for the grid level k for the region *interpolate* is not sufficient to improve the tuning model for different input sizes. In addition to the size of the finest grid, the number of processes is also important. The more processes are used, the better the data distribute over the caches and the earlier, in terms of grid level, the application switches between memory- and compute-bound.

6.2.1.1 Tuning step 1: Default execution

In the first step, the *intraphase* plugin executes a single phase of the application to determine the objective value for the default execution, i.e., default settings of the tuning parameters as provided by the batch system, and the default values for the ATPs, specified in the ATP specification file. This stage is run as part of the first steps in which PTF gathers the program's region and rts information in the first phase of the application. The measurements for the default execution are required to evaluate the improvement in the objective value, and are used later to compare the results in the last tuning step. Finally, the objective values are stored in the RTS Database for the phase and the rts's of the significant regions.

6.2.1.2 Tuning step 2: ATP tuning

The *intraphase* plugin also tunes ATPs to choose between different decomposition algorithms, preconditioners or solvers. The aim of using ATPs is to exploit the possibility to switch between the different implementations or, in a more general sense, the ability to choose between code-level alternatives at runtime. To enable ATP tuning, the application developer needs to pass the details about the tuning parameters on to the tuning system by annotating the source code at specific locations. Listing 6.7 illustrates how the ATP library API functions can be used for two dummy functions *foo* and *bar* to declare control variables from the code as ATPs. In lines 4 and 24, `ATP_PARAM_DECLARE` is used to declare the parameter names, namely `solver` and `mesh`, their types, default values and the domain. Then, `ATP_PARAM_ADD_VALUES` in lines 6 and 26 provide the possible values the parameter can take. During the experiments in DTA and RAT, ATP values have to be assigned to the corresponding control variable. The `ATP_PARAM_GET` function in lines 7 and 27 makes the mapping between the declared ATP and the corresponding control variable. The `ATP_EXPLORATION_DECLARE` macro in line 18 holds a list of search methods to apply on the domain.

An application source code may contain several ATPs, ideally independent from each other, thus enabling independent tuning. In practice, this is not always the case as the values of a parameter may depend on those of another one. Thus, the application developer may indicate the dependence by expressing them mathematically in the form of logical constraints to group dependent ATPs into domains in order to create valid configurations of the tuples of parameters. Line 28 declares a constraint between the ATPs `solver` and `mesh`.

Listing 6.7: Domain knowledge specification for ATP exploration using the ATP library.

```
void foo(){
    int atp_cv;
    ...
    ATP_PARAM_DECLARE("solver", RANGE, 1, "DOM1");
    int32_t solver_values[2] = {1,5};
    ATP_ADD_VALUES("solver", solver_values, 2, "DOM1");
    ATP_PARAM_GET("solver", &atp_cv, "DOM1");

    switch (atp_cv){
        case 1:
        // choose solver 1
        break;
        case 2:
        // choose solver 2
        break;
    }
    int32_t hint_array = {GENETIC, RANDOM};
    ATP_EXPLORATION_DECLARE(hint_array, "DOM1");
}

void bar(){
    int atp_ms;
    ...
    ATP_PARAM_DECLARE("mesh", RANGE, 40, "DOM1");
    int32_t mesh_values[2] = {0,80};
    ATP_ADD_VALUES("mesh", mesh_values, 2, "DOM1");
    ATP_PARAM_GET("mesh", &atp_ms, "DOM1");
    ATP_CONSTRAINT_DECLARE("const1", "(solver = 1 && 0 <= mesh <= 40) || (solver = 2
     ↪  && 40 <= mesh <= 80)", "DOM1");
    if((atp_ms > 1) && (atp_ms <= 40)){
    // choose mesh size 1
    }
    if((atp_ms > 40) && (atp_ms <= 80)){
    // choose mesh size 2
    }
}
```

If ATPs are specified in the application, the *intraphase* plugin tunes them first, since these typically select algorithmic alternatives, and the selection is independent of the more fine-grained tuning provided by system parameters. If there is no ATP specification, the tuning plugin directly performs the third tuning step, described in Section 6.2.1.3.

The ATP tuning step involves two components: the ATP server and the ATP library. In the first application phase, the ATP library, which was implemented by Intel generates an ATP configuration file in the JSON format that specifies the ATPs with their domains and their constraints. The *intraphase* tuning plugin reads the ATPs from the ATP description file and starts the ATP server to request for the valid points for each ATP.

The generation of a list of valid points for the parameters requires the resolution of the constraints held between parameters, and is solved using a third party constraint solver software called the Omega Calculator [99]. The Omega Calculator is an affine functions constraint solver, and generates the valid values for each ATP for the recorded constraints by filtering out the values that do not satisfy the constraints. The Omega Calculator software is composed of the Omega Library, which constitutes the core of the solver, as well as a text interpreter to query the library. One big advantage of using the Omega Calculator is the small computational time needed to solve affine function based constraints, which makes it fit for use to solve the constraints at runtime [97].

The plugin then generates the search space of ATPs using either the exhaustive or individual search strategy. This may be set by the user in the configuration file before starting PTF. The search space can consist of a single or multiple ATP domains. The exhaustive strategy for ATPs explores all valid points of the ATPs by creating a search space with the cross-product of the points. In each experiment, the ATP library receives the valid ATP settings, and assigns the values to the control variable. The individual strategy for ATPs tunes the domains individually. It starts with the first domain and evaluates all valid points for this domain, and fixes the best point for this domain. It then explores the next ATP domain until all the domains are handled. Finally, at the end of the first tuning step, we obtain an optimal configuration for the ATPs.

6.2.1.3 Tuning step 3: System parameters tuning

The third tuning step explores the system software tuning parameters, and determines an optimal system configuration. In this step, the optimal configuration for the ATPs obtained in the previous tuning step is applied. The search strategy for this tuning step is specified in the configuration file. If no strategy is specified, the individual search algorithm is selected by default. The plugin generates the search space based on the ranges of the tuning parameters it reads during initialization, and explores a single system configuration in an experiment.

During each experiment, the plugin sends rts tuning requests for the phases to set the system configuration for the entire phase, causing all the rts's to be executed with the same configuration. The plugin then sends measurement requests for the performance counters as well as the objective functions for the phase and the rts's. Individual measurements are then collected at the analysis agents, which compute the node energy for each rts. The measurements are serialized as properties and sent to the frontend, where they are deserialized.

At the end of this tuning step, the objective values for all the tested system configurations are evaluated to determine the static-best system configuration. Afterwards, the tuning results for individual rts's are evaluated to compute the optimum rts-best configurations. The combination of the best static configuration and the rts-specific best configurations is finally encapsulated in the tuning model.

6.2.1.4 Tuning step 4: Verification

The verification step is the last tuning step, and evaluates the tuning success. It uses the objective values obtained in the first tuning step for the default execution as the basis for the evaluation to determine if the best configurations that were selected for the phase and the rts's are valid. It configures the RRL with the best configurations, and performs experiments to evaluate the stability of the results. During the experiments, the plugin sends tuning requests for the phase to switch to the phase-best configuration, as well as for the individual rts's to switch to rts-best configurations. Thus, the RRL performs dynamic switching by switching to the static-best configuration at the start of the phase when the *phase enter* event is called, and to the rts-best configurations when the *region enter* event is called. If an rts of an insignificant region is encountered, the RRL sets the system configuration to the default setting.

The experiments are repeated three times to check for any variation in the results due to hardware variations or phase differences. Finally, the plugin outputs the theoretical savings characterizing the tuning result as the improvement due to static and dynamic tuning. It determines the savings for the rts's and for the whole phase using the following three definitions:

1. **Static savings for the rts's:** The total improvement in the objective value when the rts's are executed with the static-best configuration compared to the objective value for the default setting of the tuning parameters.

$$S_{RTS}^{static} = \frac{\sum\limits_{rts \in RTS} (obj_{rts}^{default} - obj_{rts}^{static})}{\sum\limits_{rts \in RTS} obj_{rts}^{default}} * 100 \tag{6.7}$$

2. **Dynamic savings for the rts's:** The total improvement in the objective value for the individual rts-specific best configurations compared to the objective value for the static-best configuration, i.e., best settings for the phase.

$$S_{RTS}^{dynamic} = \frac{\sum\limits_{rts \in RTS} (obj_{rts}^{static} - obj_{rts}^{opt})}{\sum\limits_{rts \in RTS} obj_{rts}^{static}} * 100 \tag{6.8}$$

3. **Static savings for the whole phase:** The total improvement in the objective value for the static-best configuration of the phase over the objective value for the default setting of the tuning parameters.

$$S_{RTS}^{static} = \frac{(obj_{phase}^{default} - obj_{phase}^{opt})}{obj_{phase}^{default}} * 100 \tag{6.9}$$

where,

$obj_{phase}^{default}$	=	objective value of the phase under the default configuration
obj_{phase}^{opt}	=	objective value of the phase under the static-best configuration
$obj_{rts}^{default}$	=	objective value of the rts under the default configuration
obj_{rts}^{static}	=	objective value of the rts under the static-best configuration
obj_{rts}^{opt}	=	objective value of the rts under the rts-specific optimal configuration

After the execution of the plugin, the tuning model is generated (see Section 6.2.3). The entire tuning process, i.e., the four tuning steps is repeated if further inputs are provided in the input specification file. Individual tuning models are created for each application input, and finally merged together to create the final application tuning model.

6.2.2 Tuning Inter-Phase Dynamism

PTF performs inter-phase dynamism tuning by executing the *interphase* plugin when there are changes in the dynamic characteristics across the sequence of phases. The *interphase* plugin executes four tuning steps:

default execution, cluster analysis, selective tuning and verification, as illustrated in Figure 6.2. First, the plugin executes a subset of the overall phases of the application using the default settings of the tuning parameters. Next, it executes experiments using randomly picked configurations of the tuning parameters for the selected phases, and clusters similarly behaving phases into clusters. To improve the tuning results, it performs a selective tuning step by evaluating configurations using a probabilistic random search based on previous good configurations. It then selects cluster-best phase configurations and rts-specific best configurations. Finally, the plugin checks for tuning success by executing the phases with their cluster-best configurations and the rts's with the individual rts-specific best configurations to verify whether the actual energy savings are close to the computed theoretical savings while taking into account the switching overhead. The tuning steps are described in detail in Sections 6.2.2.1- 6.2.2.4.

The plugin not only determines optimal configurations for different rts's of the code regions depending on the execution context by exploiting intra-phase dynamism, it also determines different best configurations for variations in the characteristics of the phases over time. The novel features of the *interphase* plugin are the exploitation of phase behavior to cluster similarly behaving phases into groups that have the same optimal cluster-best configuration, search space optimization using a probabilistic random search to improve the confidence of the tuning, and finally, verification of the theoretical savings with the actual savings.

To identify phases with different characteristics or behavior, phase identifiers are specified using domain knowledge, and are used in the tuning model to distinguish rts's based on the variation in the phase behavior. Phases that have similar characteristics can be grouped together into a single cluster using phase identifiers, as described in Section 5.1. This enables the selection of different configurations for rts's in different clusters. Thus, the *interphase* plugin takes into account these dynamic characteristics to determine the optimal settings, as opposed to the *intraphase* plugin, which ignores similarities in phase behavior, and exploits only the variations among the rts's of the significant regions.

Phase identifiers are provided in the same way as region identifiers via Score-P user parameters that are attached to the phase region, and are described in Section 6.3. Phase identifiers should be chosen carefully as they decide which phases are similar, and thus, have a high impact in selecting the cluster-best configurations. Typically, these should be provided by the application expert who knows how the application behaves when certain aspects in the code are modified.

6.2.2.1 Tuning Step 1 : Default Tuning

In the first tuning step, the *interphase* plugin reads the number of *samples* from the configuration file. The number of samples determines the number of experiments that will be executed, where each experiment evaluates a single phase of the application. Each experiment measures the objective value for the default configuration of the tuning parameters provided by the batch system. In each experiment, the plugin sends an rts tuning request for the phase, and issues measurement requests for the objective values as well as hardware performance counters for the phase and the rts's. The measurements for the individual rts's are then stored in the RTS Database.

It should be noted that at the beginning of the PTF run, a single phase of the application is first executed to collect the significant region information. Sometimes, not all the significant regions are encountered in the first phase, resulting in incomplete static information for the missing regions. Hence, we extended the frontend to store additional significant regions encountered during subsequent phases in the RTS Database. Once the new region information is stored in the RTS Database, subsequent experiments can then issue measurement requests for the rts's of the newly encountered significant region. The measurements in the RTS Database collected for this tuning step are used to compute the energy savings in the final tuning step.

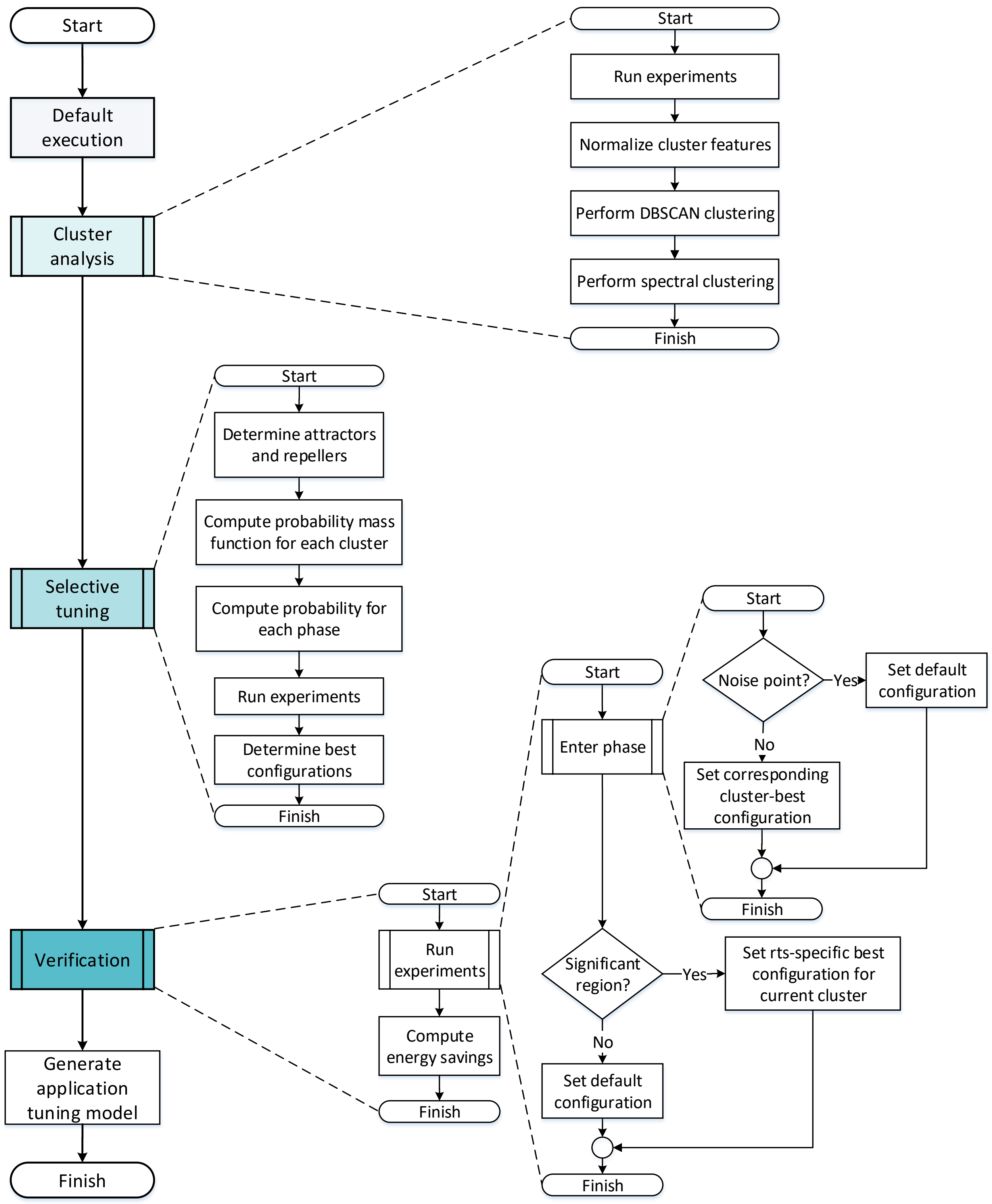

Start
Default execution
Cluster analysis
Selective tuning
Verification
Generate application tuning model
Finish
Start
Run experiments
Normalize cluster features
Perform DBSCAN clustering
Perform spectral clustering
Finish
Start
Determine attractors and repellers
Compute probability mass function for each cluster
Compute probability for each phase
Run experiments
Determine best configurations
Finish
Start
Run experiments
Compute energy savings
Finish
Start
Enter phase
Significant region?
Yes
Set rts-specific best configuration for current cluster
No
Set default configuration
Finish
Start
Noise point?
Yes
Set default configuration
No
Set corresponding cluster-best configuration
Finish

Figure 6.2: The workflow of the *interphase* tuning plugin, consisting of four tuning steps: default execution, cluster analysis, selective tuning, and verification. The plugin generates the application tuning model at the end of the tuning steps.

6.2.2.2 Tuning Step 2 : Cluster Analysis

At the start of the second tuning step, the plugin restarts the application, and creates the same number of experiments as in the previous tuning step. The plugin sends rts tuning requests for the phase to test the effect of a randomly selected configuration on the application execution. It also sends measurement requests for the objective values for the phase and the rts's. Additionally, the plugin collects hardware performance counters that are used to characterize the phases and perform clustering. Section 5.1 describes in detail the PAPI [13] events that were used to collect the hardware counters. The counters are then converted into features that are used for clustering by DBSCAN and spectral clustering, as described in Sections 5.1.1.1 and 5.1.1.2 respectively.

6.2.2.3 Tuning Step 3 : Selective Tuning

The previous tuning step evaluates only a subset of the entire search space of tuning parameters. This means that in the best-case scenario, each phase in a cluster is executed with a different configuration of tuning parameters. In the worst-case, all the phases in the cluster are executed with the same configuration. This makes it difficult to select the best configuration for the cluster with a high confidence due to the absence of other configurations for comparison.

To overcome this problem, we implemented a new tuning step that evaluates each phase in a cluster with a configuration that has never been executed for any phase in that cluster by randomly picking a configuration based on the Gaussian distribution. The idea is that the chances of picking a configuration near a good configuration that results in a lower normalized consumption are higher than picking a configuration near a worse configuration that results in a higher normalized energy.

In this tuning step, the plugin restarts the application and then optimizes the search space, as described in Section 5.2 using Algorithms 5 and 6. The plugin then requests for the objective value and the performance counters in each experiment similar to the previous tuning steps. After executing all the experiments, the plugin computes the cluster-best configuration, as well as rts-specific best configurations for each rts called in the phases of the cluster. This way, the plugin not only tunes the inter-phase dynamism, but also the intra-phase dynamism arising from changes in the behaviour of the rts's. The best configurations are chosen such that the normalized objective value, for example, $\frac{Energy}{\#AVX\ instructions}$ is minimized. This allows the plugin to tune phases with different amounts of work but of the same kind, such as more iterations of an iterative solver so that the energy per instruction is minimized.

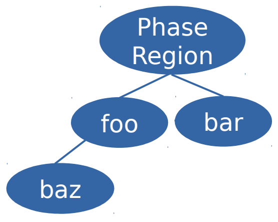
6.2.2.4 Tuning Step 4 : Verification

Similar to the *intraphase* plugin, the *interphase* plugin performs the verification step as the last tuning step to evaluate the tuning success. It uses the objective values for the phase and the rts's obtained during the first tuning step as the basis for the evaluation to determine if the optimum configurations are valid. In each experiment, the plugin sends rts tuning requests to the RRL to switch to the corresponding cluster-best configuration at the start of a new phase when the *phase enter* event is called. Similarly, the configurations for the rts's are switched to the rts-specific best configurations for the current cluster when the *region enter* event is called.

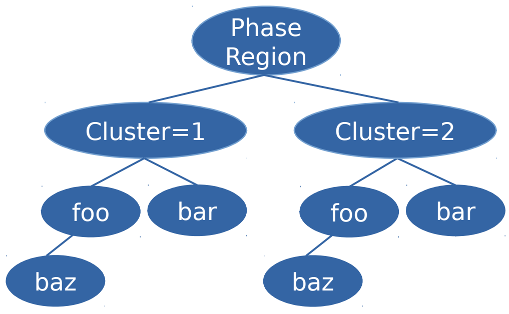
If a phase was marked as a noise point in the cluster analysis step, it is not present in any cluster, and is thus executed using the default system configuration. Similarly, if an rts of an insignificant region, i.e., an invalid rts is encountered, it is run with the default setting. The runtime system thus performs dynamic tuning by enforcing the static phase configuration for the phase, and switching system configurations for individual rts's. The plugin sends measurement requests after switching the configurations to measure the effect of inter-phase tuning and verify whether the theoretical savings match the true savings by taking into account the switching overhead.

During production runs, phases belonging to different clusters must be identified by their cluster id by the RRL in order to switch to the right configuration during a phase enter event. Therefore, the plugin performs an additional step during DTA to add the phase identifier, i.e., the cluster id into the CCT once all the experiments are completed. The cluster id represents the phase identifier, since it distinguishes phases by establishing their membership in different clusters. To enable the selection of the right configuration at runtime, the plugin clones the CCT by creating new child nodes representing different cluster ids under the phase node, and then cloning all the children of the phase node.

Figure 6.3a shows the initial CCT of the rts's of a toy application containing significant regions *bar* and *baz*. The call-path of the phase region is represented as */PhaseRegion*. If the cluster analysis step detected two clusters, the plugin creates two new nodes under the phase node, and clones all the child nodes of the phase region, as illustrated in Figure 6.3b. After this step, the cluster id represents all the phases belonging to a particular cluster. Thus, the call-path for a phase belonging to cluster 1 is */PhaseRegion/Cluster=1*, and the call-path to the rts of *bar* called from this phase is */PhaseRegion/Cluster=1/bar*.



(a) Before cloning.



(b) After cloning.

Figure 6.3: Calling Context Tree (CCT) for a toy application consisting of two significant regions, bar and baz.

The plugin then stores the tuning results, including the objective values, and the number of instances of a region executed in a run for each node. The cluster-best configuration is stored in the cluster node for the group of phases belonging to it. Additionally, each cluster node stores the phase identifier data, such as the cluster id, the phases belonging to that cluster, and the ranges of the features, i.e., the maximum and minimum value for the cluster. Storing all the information in a single node has the advantage of reducing the memory overhead because a single cluster node can be used to represent the tuning result for all the phases belonging to it. The child rts nodes store the rts-specific best configurations.

Finally, the plugin outputs the theoretical savings characterizing the tuning result as the reduction in the normalized energy consumption due to static and dynamic tuning using the following four definitions [121]:

1. **Static savings for the whole phase:** The improvement in the normalized objective value for the static-best configuration for the phase compared to the normalized objective value for the default

configuration, accumulated over all the clusters.

$$S_{phase}^{static} = \frac{\sum_{cluster=1}^{n} (obj_{phase,cluster}^{default} - obj_{phase,cluster}^{opt})}{\sum_{cluster=1}^{n} obj_{phase,cluster}^{default}} * 100 \tag{6.10}$$

2. **Static savings for the rts's:** The improvement in the normalized objective value for the rts's for the static-best setting for the cluster compared to the normalized objective value for the rts's for the default configuration, accumulated over all the clusters.

$$S_{RTS}^{static} = \frac{\sum_{cluster=1}^{n} \sum_{rts \in RTS} (obj_{rts,cluster}^{default} - obj_{rts,cluster}^{static})}{\sum_{cluster=1}^{n} \sum_{rts \in RTS} obj_{rts,cluster}^{default}} * 100 \tag{6.11}$$

3. **Dynamic savings for the rts's (w.r.t. static-best configuration):** The improvement in the normalized objective value for rts-specific best configuration over the normalized objective value for rts's for the static-best (cluster-best) configuration for the cluster, accumulated over all the clusters.

$$S_{RTS}^{dyn_1} = \frac{\sum_{cluster=1}^{n} \sum_{rts \in RTS} (obj_{rts,cluster}^{static} - obj_{rts,cluster}^{opt})}{\sum_{cluster=1}^{n} \sum_{rts \in RTS} obj_{rts,cluster}^{static}} * 100$$

4. **Dynamic savings for the rts's (w.r.t. default configuration):** The improvement in the normalized objective value for rts-specific best configuration over the normalized objective value for the default configuration accumulated over all the clusters.

$$S_{RTS}^{dyn_2} = \frac{\sum_{cluster=1}^{n} \sum_{rts \in RTS} (obj_{rts,cluster}^{default} - obj_{rts,cluster}^{opt})}{\sum_{cluster=1}^{n} \sum_{rts \in RTS} obj_{rts,cluster}^{default}} * 100$$

where,

$obj_{phase,cluster}^{default}$	=	objective value of the phase for the default configuration for a cluster
$obj_{phase,cluster}^{opt}$	=	objective value of the phase for the cluster-best configuration
$obj_{rts,cluster}^{default}$	=	objective value of the rts for the default configuration for a cluster
$obj_{rts,cluster}^{static}$	=	objective value of the rts for the cluster-best configuration
$obj_{rts,cluster}^{opt}$	=	objective value of the rts for the rts-specific optimal configuration for a cluster

6.2.3 Generating the Application Tuning Model

After the execution of the *intraphase* or the *interphase* plugin, the frontend initiates the process of tuning model generation, which was implemented by NTNU to store the knowledge from the RTS database. The tuning model helps guide the RRL in dynamic configuration switching for the phase and the rts's during production runs. The tuning model encapsulates the knowledge from DTA about the best found system configurations via scenarios, classifiers, and selectors. The classifier groups rts's with similar or identical best configurations into scenarios using a similarity score that is computed by aggregating the closeness of system configurations for a particular objective. Clustering is used to reduce the size of the tuning model by limiting the number of scenarios, and thus reduce the associated runtime overhead. A selector then returns the best configuration for each scenario for the specified objective.

The tuning model stores slightly different information for intra-phase and inter-phase tuning. Intra-phase tuning using the *intraphase* tuning plugin not only tunes the application for a single input, but also tunes ATPs and learns from running the application for different inputs. Rts's with different characteristics in different application runs are distinguished with the help of input identifiers, such as the grid size, as described in Section 6.2.1. For each input provided in the input specification file, the entire tuning process, i.e., the four tuning steps in the *intraphase* plugin is repeated, and individual tuning models are created for each application input. A tuning model merger [122] deserializes the individual tuning models and extracts all tuning information, such as rts's and their best configurations as well as the corresponding input identifiers. It filters out all duplicated rts's, and produces a new final set of scenarios by clustering rts's with similar configurations, and then serializes the merged tuning model information into a single application tuning model in the JSON format.

The clustering mechanism clusters rts-best configurations using hierarchical clustering, and performs three steps [14]: dendrogram generation, cluster generation, and scenario creation. The first step generates a dendrogram, i.e., a tree expressing the similarity of rts's based on the distances between their system configurations. To perform clustering, the algorithm first treats every rts as a cluster with a single element, i.e., the rts itself. It uses the Lance-Williams algorithm to recursively compute the inter-cluster distance to merge the closest clusters, and then recompute the distance of the newly created cluster to all other clusters and merge the next closest clusters until all clusters are merged into a single cluster. The cluster generation uses the dendrogram to create a clustering of the rts's by performing a tree cut by partitioning the tree's nodes into disjoint subsets such that it minimizes the dispersion of data within clusters, while maximizing the dispersion between the clusters. It should be noted that the clusters of rts's here do not refer to the cluster analysis results of the inter-phase tuning. The third step groups the clusters into scenarios, and creates a selector that returns an optimal system configuration for the scenario. This system configuration is then used for all rts's in the cluster at runtime.

For inter-phase tuning, the tuning model generation clusters rts's with identical configurations into scenarios with unique ids. First, it iterates over the phases as well as the rts's from the RTS Database, and creates a new scenario for each cluster-best and rts-best configuration. If another scenario with an identical configuration exists, the two scenarios are merged. Additionally, the tuning model stores information about the clusters generated during DTA, including the cluster ids, the phases belonging to the corresponding clusters, the cluster features and their ranges that were used for clustering by requesting the information from the RTS Database. This additional information is then used by the RRL during production runs to identify the cluster id of an executing phase.

6.3 Runtime Application Tuning

Runtime Application Tuning (RAT), controlled by the Runtime Management module, starts by initializing the application and the underlying system software, including the measurement infrastructure and the RRL. The Runtime Management module instantiates the TMM at the beginning of the application execution. The TMM reads and deserializes the tuning model generated at the end DTA, and stores the information as hash maps for efficient look up at runtime. The Runtime Management module was implemented by TUD, and the TMM was implemented by TUD and NTNU.

When an enter region event is called from Score-P during the application run, the Runtime Management module maintains the current call stack and collects additional identifiers, such as the input parameters, phase identifiers or hints/domain knowledge provided by the application expert in the form of user parameters. It constructs the full call-path for every rts using the total set of context elements, i.e., input, phase, and significant region information. The TMM then checks if this region is found in the tuning model, and if found, marks the region as significant. If the region is not found, it is marked as an unknown region. For each significant region, the RTS Management module sends the complete call-path together with the additional identifiers to the TMM, and requests the best configuration. The TMM returns a configuration, which is stored in a hash map using the call-path and the additional identifier information as the key. Subsequent calls to an rts simply involve a lookup of the scenario id for the rts in the hash map.

The returned configuration is passed to the Parameter Controller, which sets the configuration through dedicated Parameter Control Plugins (PCPs) for each tuning parameter. PCPs are also used during DTA in the *system parameters tuning* and *verification* tuning steps of the *intraphase* plugin, and in the *cluster analysis* and *verification* tuning steps of the *interphase* plugin. The PCPs for DVFS and UFS support x86_adapt [65], msr-safe, LIKWID and sysfs. The Parameter Controller maintains a configuration stack that stores the configurations returned by the TMM in one of the following two modes using the environment variable `SCOREP_RRL_CHECK_IF_RESET`:

- **`reset`**: In the `reset` mode, every configuration of the tuning parameters is pushed onto the configuration stack. When the region exit event is called, the current configuration is unset by popping it from the stack, and the previous configuration from the top of the stack is set. The `reset` mode is enabled by default.
- **`no_reset`**: In the `no_reset` mode, only the default configuration and the current configuration of the tuning parameters will be saved on the stack. Each configuration stays active until a new configuration is set, which overwrites the current configuration in the stack.

If input identifiers were specified for intra-phase tuning, the name of the input identifiers specification file is exported for RRL using the environment variable `SCOREP_SUBSTRATE_RRL_INPUT_IDENTIFIER_SPEC_FILE`. If the environment variable is not set, the tuning stops with an error message. The RRL also uses the ATP library during RAT to set the best configuration of the ATPs for the current input identifier fromthe merged tuning model upon encountering a valid rts. The ATP library assigns the ATP value to the control variable defined in the application source code. In case no configuration is available for the current input set, the default parameter settings are assigned, both for the ATPs and the other tuning parameters.

For inter-phase tuning, RRL uses the cluster id returned by the cluster prediction library, which is linked to the application. The cluster number of a phase is used as a phase identifier to express the combination of the cluster features that were used to group the phases based on their similarities during DTA. For runtime tuning, the cluster id is added to C/C++ and Fortran applications as the phase identifier by defining it as a user parameter. This way, the cluster number is added to the call-paths of the rts's when they are called from the phase. Phase identifiers are annotated using a Score-P user parameter by first declaring

Listing 6.8: Score-P phase identifier specification for runtime cluster prediction using the cluster prediction library for Fortran applications.

```fortran
#include "scorep/SCOREP_User.inc"
...
SCOREP_USER_REGION_DEFINE(R1)
SCOREP_USER_PARAMETER_DEFINE(param)
...
do it = 1, max_iter
! Phase region begins
SCOREP_USER_OA_PHASE_BEGIN(R1,"PhaseRegion",0)
SCOREP_USER_PARAMETER_INT64(param, "Cluster", predict_cluster())
...
SCOREP_USER_OA_PHASE_END(R1)
end do
```

a unique handle using a `SCOREP_USER_PARAMETER_DEFINE(handle)` macro, and then the defining the user parameter via `SCOREP_USER_PARAMETER_INT64(handle, "Cluster", predict cluster())`. For Fortran applications, the definition requires the Score-P handle as the first argument, followed by the name and the value of the user parameter. Listing 6.8 presents the phase identifier specification for Fortran applications.

For C/C++ applications, the header file `cluster_predictor.h` is included in the file containing the phase region. The phase identifier is defined directly using `SCOREP_USER_PARAMETER_INT64("Cluster", predict cluster())` that requires only the parameter name and the value, as shown in Listing 6.9. The user parameter definition for both C/C++ and Fortran applications immediately follows the OA phase region annotation. Both application types are then linked to the cluster prediction library by adding the linker flags in the Makefile. The user parameter definition sets the parameter name to *Cluster*, and calls the cluster prediction library by invoking the *predict_cluster()* function. The cluster prediction library predicts and returns the cluster number of the current phase, which is then set as the value of the user parameter *Cluster*.

Listing 6.9: Score-P phase identifier specification for runtime cluster prediction using the cluster prediction library for C/C++ applications.

```c
#include <scorep/SCOREP_User.h>
#include <cluster_predictor.h>
...
SCOREP_USER_REGION_DEFINE(R1)
...
for (it = 1; it < max_iter; it++) {
    // Phase region begins
    SCOREP_USER_OA_PHASE_BEGIN(R1,"PhaseRegion",0)
    SCOREP_USER_PARAMETER_INT64("Cluster", predict_cluster())
    ...
    SCOREP_USER_OA_PHASE_END(R1)
}
```

The *predict_cluster()* method calls the cluster prediction library, which is implemented as a Singleton class so that only a single instance can ever exist. Section 5.3 describes the library in detail. When the application enters the phase region, i.e., it enters the main progress loop, the *enter phase* event handler is executed. When the first phase is executed, the instance of the cluster prediction class is created, and the *is_initialized* flag is set. The library requests for the ranges of the cluster features, and also the set of all phases belonging to the clusters from the TMM. It initializes the PAPI library once at the beginning of the first phase. It then creates an event set using the PAPI low-level API to request the performance counters for the AVX calculation instructions, L3 cache misses, the conditional branch instructions, and L2 cache misses for every phase.

For each phase, the *predict_cluster()* function calls the *predict()* method. If the phase was evaluated during DTA, the method simply returns the corresponding cluster number, which sets the *value* of the user parameter. The Runtime Management module requests the PCPs to switch to the corresponding cluster-best configuration for the current phase. For rts's of the significant regions called inside the current phase, the configuration switching is handled as usual by RRL by looking up the configuration for the current rts using its call-path and additional region identifiers. If the current phase is not found in the cluster information, the *predict()* method assumes that the current phase was designated as a noise point during DTA, and the PCPs switch to the default configuration for the phase and all the rts's called inside it.

For the phases that were not seen during DTA, the *predict()* method predicts the cluster number of the current phase based on one of the three cluster predictors, detailed in Sections 5.3.1 and 5.3.2. The configuration for the phase as well as the rts's are switched dynamically by the PCPs based on the cluster id returned by the *predict_cluster()* method. An additional step then corrects possible mispredictions of the cluster number for the previous phase. First, the PAPI counters for the previous phase are converted into features, and compared against the ranges of the cluster features of the predicted cluster. If the features fall within the ranges of another cluster, the cluster number of the previous phase is corrected, so that future predictions can use the updated cluster information. If the values of the cluster features are not in the ranges of any known cluster, the phase is assigned as a noise point, and the PCPs switch to the system default configuration for the current phase and all the rts's.

6.3.1 Calibration

RAT distinguishes between seen and unseen rts's of the significant regions. Unseen rts's might occur for several reasons, for example, when the significant region is already known, but the application is run with different parameters or inputs during RAT, or when a significant region that was never encountered during DTA is encountered during RAT. For unseen rts's, RAT can be configured to perform calibration, which was implemented by TUD, and trains an Artificial Neural Network (ANN) to predict optimal configurations at runtime.

For intra-phase tuning, RAT performs calibration for the unseen rts's if the calibration module is enabled. For inter-phase tuning, RAT can take one of two approaches: perform either runtime cluster prediction for the unseen phases, or calibration for the unseen rts's. In the first approach, the application is linked with the cluster prediction library and the calibration is turned off, during which RAT invokes the cluster prediction function to perform runtime prediction of the cluster ids for unseen phases, as described in Section 5.3. During this process, if an unseen rts of a significant region is encountered, the PCPs switch to the default system configuration for the rts since it is not found in the tuning model. In the second approach, the application is linked to the cluster prediction library and calibration is turned on. First, the cluster predictor predicts the cluster id for the current phase, and the PCPs switch to the corresponding cluster-best configuration. Then, if an unseen rts of a significant region is encountered during the current phase, the calibration mechanism is invoked. The calibration module selects an optimal configuration for the rts, which is applied each time this rts is encountered, regardless of the cluster id of the executing phase. This is ineffective, since calibration ignores the similarities between the phases, and selects a single optimal configuration for the rts across all the clusters.

During calibration, the objective values for all the significant regions are collected, and then filtered out for regions whose execution time is less than 100 ms. Calibration then creates a state matrix of the product of the tuning parameters, for example, of size {core frequencies x uncore frequencies}, where a state is a pair of core and uncore frequencies. When RRL encounters an unknown rts, the algorithm first starts at a certain state, and starts measuring the energy consumption. It computes the Q-value for each configuration when a change to a new frequency is requested. Every change to a different frequency is considered as an action, and is performed by selecting the next configuration from the direct neighbors of the current configuration based on the Q-Value and the learning rate. An action is taken if the cost/Q-value is smaller than the previous value. If the cost is the lowest for a certain state, it is selected as the optimal configuration. The algorithm finally terminates when the program has finished executing.

6.4 Summary

In this chapter, we presented the integration of our work into the two stages of the tuning methodology, namely DTA and RAT. We also defined the interactions between different components of the architecture during the two stages. First, pre-analysis steps are performed using two automatic tools, namely *scorep-autofilter* and *readex-dyn-detect* to prepare the application for tuning. To reduce the measurement overhead of executing many fine-granular program regions, *scorep-autofilter* measures the granularity of program regions and generates a Score-P filter file containing the list of frequently executed regions with a very short execution time per instance. Score-P then omits the instrumentation for these regions, thus reducing the instrumentation overhead. From the remaining program regions, *readex-dyn-detect* selects significant regions that are worth tuning by performing a dynamism analysis w.r.t. variations in the execution time for individual rts's and between phases, and in the compute intensity between significant regions. Dynamic tuning is aborted if insufficient tuning potential is found, and regular static tuning is applied instead.

We also present application experts with the opportunity to specify domain knowledge for defining parameters in the form of identifiers for the input, phase and user regions to expose the dynamic behaviour of the application. The identifiers allow to distinguish different system scenarios and improve the tuning results. Moreover, the domain knowledge specification also exposes ATPs such as preconditioners or domain decomposition methods to the tuning process. For a fine-grained analysis and tuning, significant regions are instrumented by inserting probe functions around the relevant code regions.

DTA is performed by PTF using two plugins, namely *intraphase* plugin for intra-phase tuning, and *interphase* plugin for inter-phase tuning. PTF reads the configuration file, and calls a tuning plugin, which performs one or more tuning steps in which experiments are executed to measure the effect of the system configuration on the objective. Both plugins perform DVFS, UFS and DCT using four tuning steps. However, they use different approaches for DTA. The *intraphase* tuning plugin finds the best system configuration for the rts's while ignoring the characteristic behaviour of the application phases. For each application input, it executes the phase with the default system configuration, tunes the ATPs, followed by the hardware and software tuning parameters, and finally, verifies the tuning success. The *interphase* tuning plugin first executes the phases with the default system configuration, and then executes experiments to evaluate system configurations using a random strategy. It clusters similarly behaving phases, and then evaluates more configurations using a selective tuning step based on the Gaussian distribution. Finally, it measures the tuning success. The two plugins restart the application at the end of each tuning step.

After all the tuning steps are completed, PTF generates the application tuning model, which stores the best configurations in the form of scenarios, classifiers, and selectors. Rts's with identical configurations are grouped into scenarios using a classifier, and a selector returns the best configuration for each scenario. For intra-phase tuning, individual tuning models are generated for each application input, and then merged to form the application tuning model. For inter-phase tuning, the tuning model stores the cluster information, such as the phases belonging to each cluster and the ranges of the cluster features, as well as cluster-best configurations and rts-specific best configurations.

The tuning model is read and deserialized at runtime to enable scenario detection and configuration switching during production runs for RAT. For intra-phase tuning, the RRL requests the Parameter Controller to set the PCPs to the static-best configuration for the phase at the *phase enter* event, and to the rts-best configuration at the *region enter* event. For inter-phase tuning, the Parameter Controller sets the PCPs for a phase to the corresponding cluster-best configuration while switching to the rts-best configurations for the rts's within the phase. Additionally, the cluster prediction library predicts the cluster id for all phases that were not seen during DTA.

In addition to the workflow of DTA and RAT, we provided a brief description of the calibration mechanism that is invoked for unseen rts's of significant regions. For both intra-phase and inter-phase tuning, calibration is performed if the calibration module is enabled. One major difference is that calibration for inter-phase tuning does not take into consideration the similarities in the characteristics between different phases, and can only predict a single common rts-best configuration for an unseen rts for all clusters. Calibration is performed by training an ANN using Q-learning that determines if a new configuration should be selected for an unseen rts based on the Q-value determined for the configuration.

7

Evaluation

In this chapter, we evaluate our tuning methodology, starting from application instrumentation, fine-granular region filtering, dynamism detection, significant region detection, DTA, tuning model generation, and finally, RAT. We focus on applications that exhibit inter-phase dynamism, and tune them using the *inter-phase* tuning plugin. Thus, we tune three real-world applications, namely 128.GAPgeofem from the SPEC MPI2007 benchmark suite, sam(oa)^2 from the Chair of Scientific Computing at the Technische Universität München, INDEED, a production application, and miniMD, a proxy benchmark from the Mantevo project.

Our methodology selects the best configuration for each cluster of phases, and for every individual rts within each cluster. We present the static savings for the phase as well as the rts's, and the dynamic savings for the rts's w.r.t. the cluster-best and the system default configurations. The best configurations for the clusters and the rts's are encapsulated in the tuning model, which guides the runtime tuning.

At runtime, each application is first linked with the runtime cluster prediction library. The RRL then reads the tuning model and dynamically switches system configurations for the phases and the rts's. We present the improvements in the job energy consumption as well as the CPU energy for each application, and compare the savings for the three cluster predictors. We also perform an overhead analysis to determine the performance loss due to the switching overhead at runtime.

We first describe the platform and system architecture that was used to perform the evaluation in Section 7.1, followed by a description of the four benchmarks in Section 7.2. Section 7.3 presents the pre-analysis steps, followed by the results of cluster analysis by DBSCAN and spectral clustering for the four applications. It also compares the dynamic energy savings w.r.t. the job energy and CPU energy, as well as the performance overhead of the three cluster predictors.

7.1 Platform Description

To test our tuning methodology, we used Technische Universität Dresden's (TUD) Top500 cluster Taurus located at Dresden in Germany. Taurus offers heterogeneous compute resources, namely 1456 Intel Haswell nodes, each with 64, 128 or 256 GB of RAM per node, 32 Intel Broadwell nodes, each with 64 GB of RAM per node, 6 large SMP nodes with 2 TB RAM, and 44 Intel Sandy Bridge CPUs with NVidia K20x

GPUs. We evaluated our approach on the Haswell partition of Taurus. The Haswell partition consists of 1456 compute nodes based on the Intel Haswell-EP architecture. Each node consists of two 12-core Xeon E5-2680 v3 sockets with Hyper-Threading and TurboBoost disabled. We ran our experiments on nodes with 64 GB of RAM per node. The core frequency of each CPU core ranges from 1.2 GHz to 2.5 GHz, while the uncore frequency of the two sockets ranges from 1.2 GHz to 3.0 GHz. Each node runs with a default CPU frequency of 2.5 GHz and an uncore frequency of 3 GHz. The CPU and uncore frequencies are switched using the low-level x86_adapt library, which is a Linux kernel module that enables logging and setting/resetting system parameters stored in MSR or PCI registers of x86 processors [65].

7.2 Benchmark Specification

In this section, we present an overview of the three real-world applications and one proxy benchmark that were used to test our approach. We used the following applications for our evaluation:

1. 128.GAPgeofem: 128.GAPgeofem is a Finite-Element Method (FEM) code to measure the transient thermal conduction, and is part of the SPEC MPI2007 benchmark suite. It is written in a mixture of C and Fortran, and runs 235 iterations of the simulation loop.
2. INDEED: INDEED is a sheet metal forming simulation software, and performs adaptive mesh refinement. It is written in Fortran, and runs 154 iterations of the simulation loop.
3. sam(oa)^2: sam(oa)^2 is an adaptive mesh refinement framework for flows in porous media and tsunami simulations. It is written in Fortran, and runs 1141 iterations of the time loop.
4. miniMD: miniMD is a parallel Molecular Dynamics (MD) code in the Mantevo project at Sandia National Laboratories, USA. It is written in C++, and runs 100 iterations of the simulation loop.

Table 7.1: Summary of the benchmarks used for evaluation.

Application	Parallelism	Language	Iterations	Type
128.GAPgeofem	MPI	C, Fortran	235	Real-world
INDEED	OpenMP	Fortran	154	Real-world
sam(oa)^2	MPI+OpenMP	Fortran	1141	Real-world
miniMD	MPI+OpenMP	C++	100	Proxy

Table 7.1 presents an overview of the four applications. 128.GAPgeofem, sam(oa)^2 and miniMD are MPI applications, while INDEED is an OpenMP application. All the applications were compiled using the Intel/2018 compiler and Intel MPI version 2018.1.163, and instrumented with Score-P version 4.0. The MPI applications were evaluated on a single node as well as multiple nodes, while INDEED was evaluated on a single node using 12 OpenMP threads.

7.3 Exploitation of Inter-Phase Dynamism

We first introduce an overview of the common steps that were performed for all the applications before starting the inter-phase analysis. We then present individual application-specific details and the results of clustering and the energy savings in the respective sections.

First, the applications were compiled by adding the Score-P instrumentation wrapper `scorep --online-access --user` to the Makefile of all the applications. The `--online-access` flag turns on the communication between Score-P and PTF via the OA interface, and the `--user` flag enables user instrumentation.

The applications were then run normally, at the end of which, Score-P generated a profile in the *cubex* format. To reduce the instrumentation overhead, we used *scorep-autofilter* to create a filter file to suppress the measurements for fine-granular instrumented regions with execution time under 10 ms.

In the next step, we used *readex-dyn-detect* to detect and analyze the dynamism of the applications. The *readex-dyn-detect* tool requires a single phase region, which is a repetitive, single-entry and exit region, typically the body of the main progress loop. To identify the phase region in each application, we first used *CUBE* to visualize the flat profile and detect the body of the simulation loop using the metric *Number of Visits*, which specifies how many times a region was called during the application run.

We then manually annotated the phase region of the applications, and exported the environment variables `SCOREP_PROFILING_FORMAT` and `SCOREP_METRIC_PAPI` as described in Listing 6.2 to obtain a tupled profile, which was used as input by *readex-dyn-detect*. We ran *readex-dyn-detect* using a granularity threshold of 100 ms to define the minimal mean execution time for regions to be considered significant for tuning. Additionally, for the detection of intra-phase dynamism, we set the minimum standard deviation of the compute intensity of the significant regions to 10%, and of the execution time of the rts's to 10%. For inter-phase dynamism, we set the minimum standard variation of the phase time in percentage of the mean phase time to 10%. The tool then identified the significant regions and the dynamism, and generated a configuration file containing this information.

In the next step of DTA, we specified the following attributes in the configuration file:

1. Tuning parameters: The *interphase* tuning plugin supports three tuning parameters: core frequency, uncore frequency and the number of OpenMP threads. In our work, we only used the core and uncore frequencies for tuning due to a bug in the Score-P OA interface, which reinitializes the entire metric subsystem at the beginning of a new phase. This causes problems in measuring PAPI counters because the spawned OpenMP threads exit after running their parallel sections before the metric subsystem is reinitialized, thus causing the failure of the initialization of the PAPI library from the second phase.

   We specified the ranges (minimum, maximum and the step size) for the core frequency in MHz as {1200, 2500, 100} and for the uncore frequency in MHz as {1200, 3000, 100}.

2. Objectives: Since our work primarily focuses on optimizing the energy-efficiency, we specified normalized energy as our tuning objective. Additionally, we also enabled the measurement of the execution time.

3. Energy plugin: We used the *x86_energy_sync_plugin* [120] to collect the energy measurements. This is a Score-P power and energy event plugin counter, and supports reading msr registers directly or through the x86_adapt library on Taurus. The plugin is a strictly synchronous plugin, and returns the energy consumption via one responsible process per node. The *x86_energy_sync_plugin* uses Intel's RAPL interface to return energy measurements for the following RAPL domains [123]:

PKG	Whole CPU package
PP0	Processor cores
PP1	Uncore devices
DRAM	Memory controller

   For our evaluation, we requested the energy consumption of the whole package.

We compared the theoretical savings obtained during DTA, and the runtime dynamic savings with the switching overhead, w.r.t. three parameters: job energy, CPU energy, and execution time. To obtain the measurements for job energy and execution time, we used the SLURM Resource and Job Management System tool *sacct* [39], which allows per-job energy accounting with power profiling capabilities, and enables users to query post-mortem data for previously executed jobs. The values of job energy and time were obtained using the `--format "ConsumedEnergyRaw,CPUTimeRAW"` command. The CPU energy was measured using *measure-rapl*, a runtime tool that was developed in the READEX project, and uses the x86_adapt library to measure the CPU energy via RAPL.

The values of job energy, CPU energy and execution time for the three cluster predictors were compared against the untuned instrumented version for each application. The runtime savings obtained are the result of the averaged measurements for two consecutive runs of the applications on the same node(s) for the single and multi-node experiments.

To exploit the inter-phase dynamism, the *interphase* tuning plugin first performs DBSCAN to cluster points that are close to each other. Then, the plugin analyzes the noise points, i.e., points that were not clustered during DBSCAN to determine associations between them using spectral clustering. It computes the normalized graph Laplacian matrix, and determines the eigengap using the eigenvalues to define the number of clusters for K-means. The *interphase* plugin then assigns the noise points to new clusters, and proceeds to the selective tuning step.

The following sections describe in detail the evaluation of DTA and RAT for 128.GAPgeofem (Section 7.3.1), sam(oa)^2 (Section 7.3.2), INDEED (Section 7.3.3) and miniMD (Section 7.3.4).

7.3.1 128.GAPgeofem

128.GAPgeofem [124] is a parallel Finite-Element Method (FEM) code from the SPEC MPI2007 benchmark suite. The SPEC MPI2007 benchmark suite is an industry-standard HPC benchmark suite that contains 18 MPI compute-intensive applications. The medium data set scales well up to 128 ranks, and the large data set scales up to 2048 ranks. 128.GAPgeofem provides both medium and large input datasets, and is written in C and Fortran. It is based on the GeoFEM (Geophysical Finite Element Methods) [125] software. GeoFEM was developed as part of a Japanese five-year project called ‘Earth Simulator project’, which forecasts various earth phenomena through the simulation of a virtual earth placed in a supercomputer by modeling solid earth phenomena such as the convection between the mantle and core, plate tectonics, seismic wave propagation and other phenomena.

GeoFEM includes parallel finite element codes for linear and non-linear solid mechanics and thermal fluid simulations, a parallel iterative linear solver library, a grid partitioning subsystem, a parallel visualization subsystem, and utilities for parallel I/O. It also allows other users to plug-in their own FEM codes to the GeoFEM platform.

GAPgeofem simulates transient thermal conductions with gap radiation, which refers to the radiative heat transfer that occurs between closely adjacent surfaces with heterogeneous material properties. The benchmark uses a backward Euler implicit time-marching scheme, a parallel CG (conjugate gradient) iterative solver, and the SSOR (symmetric successive over-relaxation) preconditioner. GAPgeofem generates three executables during the build: GAPmetis, GAPpart, and GAPgeofem, where the first two executables partition the input mesh to the number of cores in the target system, and then the third executable, GAPgeofem performs the actual FE simulation. The source files for the first step are written in C, and in Fortran for the second and third steps.

GAPgeofem uses point-to-point non-blocking MPI communications for exchanging domain boundary information, and collective MPI calls for calculating dot-products and global minimum and maximum values [126]. For our experiments, we used the medium size input dataset, which executes 235 iterations of the simulation loop, and represents an FE mesh of a section of the earth with 2 million degrees of freedom.

7.3.1.1 Evaluation of Design-Time Analysis

The phase region for 128.GAPgeofem was determined using a combination of source code inspection and visualization of the *cubex* file using CUBE, and then annotated to perform DTA. Table 7.2 presents the result of the significant region analysis and dynamism detection by *readex-dyn-detect*.

Table 7.2: Significant regions identified by *readex-dyn-detect* for 128.GAPgeofem.

Significant region	Rts call-path	Intra-phase dynamism	
		Compute intensity	**Execution time**
geofem_solver_cg_11	/PhaseRegion/geofem_solver_11/ geofem_solver_cg_11	✔	✔
mat_ass_thermal_main_361	/PhaseRegion/mat_ass_thermal/ mat_ass_thermal_main_361	✔	✘

This shows that most of the computation time in the FEM procedure is spent in two processes: assembling the coefficient matrix, defined in the region *mat_ass_thermal_main_361*, and solving the linear equations, defined in the region *geofem_solver_cg_11*. Column 1 indicates the significant regions returned by *readex-dyn-detect*, column 2 represents the instance of the significant region, i.e., its rts, which can be identified by its call-path. Columns 3 and 4 indicate whether the dynamism detected by *readex-dyn-detect* was due to the variation in the compute intensity or the execution time of the rts's. As we can see, both *geofem_solver_cg_11* and *mat_ass_thermal_main_361* show variations in the execution time, while the matrix assembly (*mat_ass_thermal_main_361*) only shows variations in the compute intensity.

For both single node and multi-node runs, we set the number of *samples*, i.e., the number experiments or iterations of the simulation loop to run during a tuning step to 80. This means that we performed cluster analysis for these 80 points during DTA.

Single node

For single node experiments, we ran 128.GAPgeofem on a single compute node on Taurus using 24 MPI processes on 24 cores.

Cluster analysis

In Figure 7.1, the points on the graph represent the 16 eigenvalues for the 16 noise points obtained as a result of DBSCAN on 128.GAPgeofem. The X-axis represents the number of eigenvalues, and the Y-axis represents the eigenvalues for the normalized graph Laplacian matrix. We can see that there are small changes between the first two eigenvalues, and a large eigengap between the second and third eigenvalues, as shown by a red line. Thus, the value *k*, representing the input for the number of clusters for K-means at the end of spectral clustering was selected as 2.

Figure 7.2 illustrates the final clustering consisting of six clusters obtained after performing DBSCAN and spectral clustering for 128.GAPgeofem. Clusters 1-4 result from DBSCAN, while clusters 5 and 6 result from spectral clustering. We can see that the individual clusters have marked demarcation majorly w.r.t. the difference in the number of normalized L2 cache misses. The best configuration of the CPU and uncore frequencies for each cluster is illustrated by the {CPU_freq, uncore_freq} setting. Cluster 5, which lies in an area of low branch instructions has the highest setting for both core and uncore frequencies. It can also be observed that all the clusters except cluster 5 have a lower value of the CPU frequency setting.

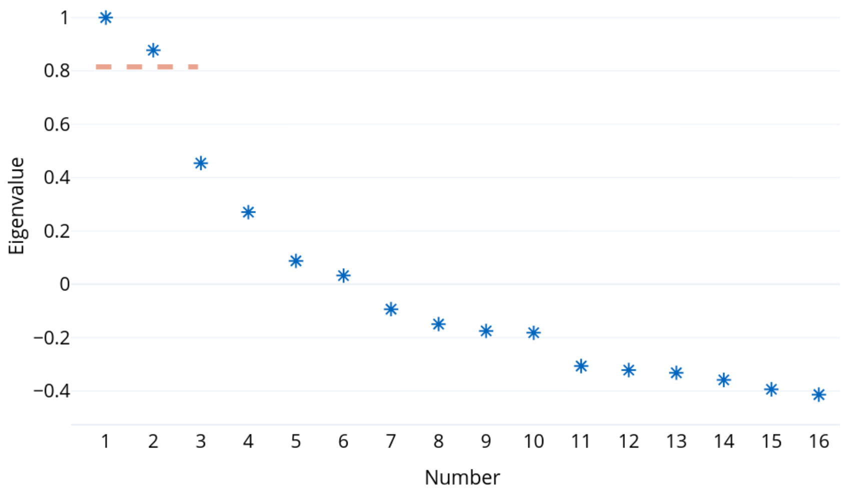


Figure 7.1: Eigenvalues computed for the graph Laplacian matrix of 128.GAPgeofem for a single node run.

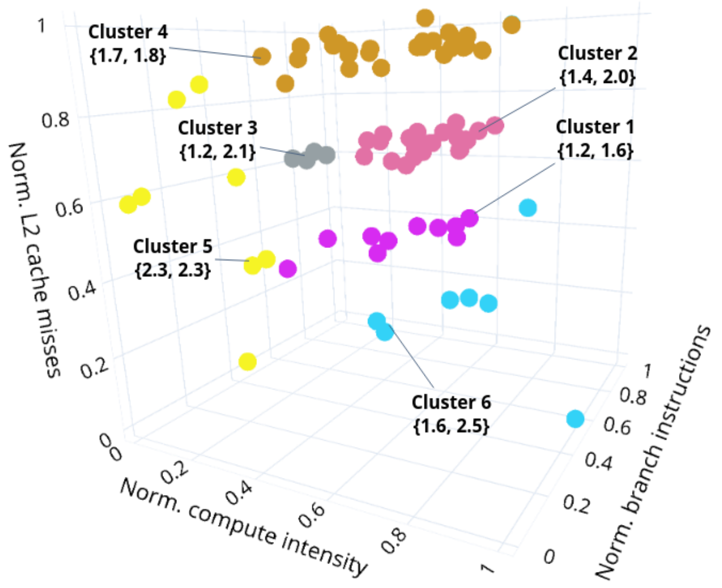


Figure 7.2: Single node results of the cluster analysis (DBSCAN followed by spectral clustering) performed on 80 phases of 128.GAPgeofem. Six clusters are produced, and the best configuration for each cluster is depicted in the form {CPU_freq, uncore_freq}.

Best rts-specific configurations

Table 7.3 lists the rts-specific best configurations for the rts's of 128.GAPgeofem for each cluster. The rts's are identified by their call-paths, and the best configurations are specified by the {CPU_freq, uncore_freq} settings. As we can see, the cluster-best configurations for the phases depicted in Figure 7.2 are also the best configurations for the rts's of *geofem_solver_cg_11*. This means that the intra-phase dynamism does not have an effect on the selection of the rts-best configurations. The best settings for the rts's of *mat_ass_thermal_main_361* indicate a high CPU frequency and low uncore frequency for all clusters except for cluster 6.

Table 7.3: Rts-specific cluster-best configurations of the tuning parameters {CPU_freq, uncore_freq} for the rts's of the significant regions of 128.GAPgeofem for a single node run.

Rts	**Clusters**					
	1	**2**	**3**	**4**	**5**	**6**
/PhaseRegion/geofem_solver_11/ geofem_solver_cg_11	{1.2, 1.6}	{1.4, 2.0}	{1.2, 2.1}	{1.7, 1.8}	{2.3, 2.3}	{1.4, 3.0}
/PhaseRegion/mat_ass_thermal/ mat_ass_thermal_main_361	{2.5, 1.7}	{2.4, 1.2}	{2.5, 1.2}	{2.5, 1.7}	{2.3, 1.7}	{2.5, 2.5}

Multiple nodes

For multi-node experiments, we ran the application on two compute nodes on Taurus using 48 MPI processes on 48 cores.

Cluster analysis

In Figure 7.3, the points on the graph represent the 21 eigenvalues for the 21 noise points obtained as a result of DBSCAN on 128.GAPgeofem. The X-axis represents the number of eigenvalues, and the Y-axis represents the eigenvalues for the normalized graph Laplacian matrix. We can see that there are small changes between the first two eigenvalues, and a large eigengap between the second and third eigenvalues, as shown by a red line. Thus, the value k was selected as 2.

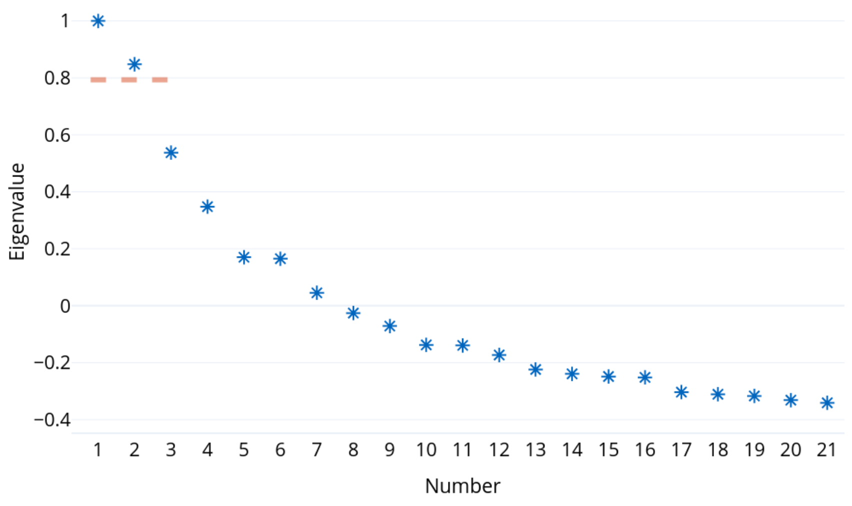


Figure 7.3: Eigenvalues computed for the graph Laplacian matrix for 128.GAPgeofem for a multi-node run.

Figure 7.4 illustrates the final clustering consisting of seven clusters obtained after performing DBSCAN and spectral clustering for 128.GAPgeofem. Clusters 1-5 result from DBSCAN, while clusters 6 and 7 result from spectral clustering. Although the clusters have a dispersion based on the level of L2 cache misses, they consist of fewer points than in the case of the single node run. The best configuration of the CPU and uncore frequencies for each cluster is defined by the {CPU_freq, uncore_freq} setting. As with the case of the single node run, it can be observed that all the clusters have a relatively low value of the CPU frequency.

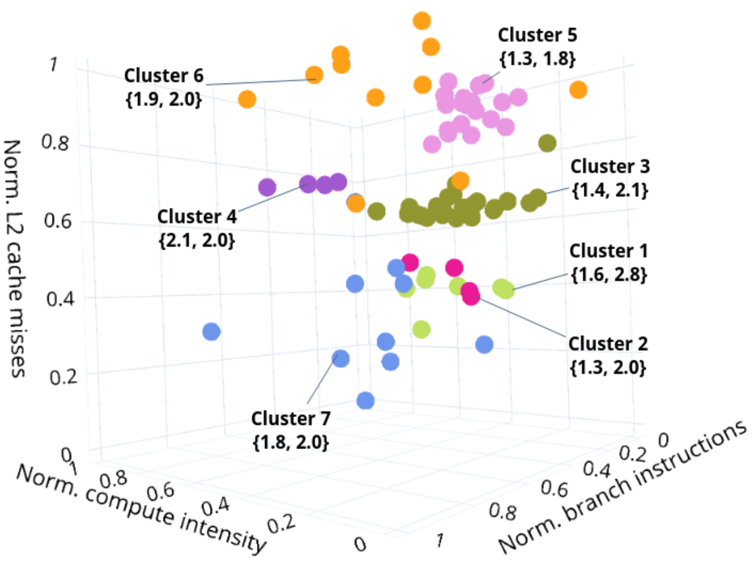


Figure 7.4: Multi-node results of the cluster analysis (DBSCAN followed by spectral clustering) performed on 80 phases of 128.GAPgeofem. Seven clusters are produced, and the best configuration for each cluster is depicted in the form {CPU_freq, uncore_freq}.

Best rts-specific configurations

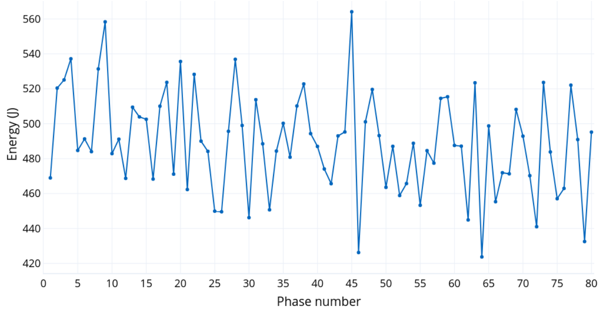
Table 7.4 lists the rts-specific best configurations for the rts's of 128.GAPgeofem for each cluster. The rts's are identified by their call-paths, and the best configurations are specified by the {CPU_freq, uncore_freq} settings. Except for clusters 4 and 7, the rts-specific best configurations for *geofem_solver_cg_11* are the same as the cluster-best configurations for the phases depicted in Figure 7.4. This shows that the intra-phase dynamism does have an effect in multi-node runs. Moreover, the CPU frequency setting for all the clusters is in a low-medium range, with a medium-high setting for the uncore frequency. The best settings for the rts's of *mat_ass_thermal_main_361* indicate a medium-high range for the CPU frequency and low uncore frequency for all the clusters.

Table 7.4: Rts-specific cluster-best configurations of the tuning parameters {CPU_freq, uncore_freq} for the rts's of the significant regions of 128.GAPgeofem for a multi-node run.

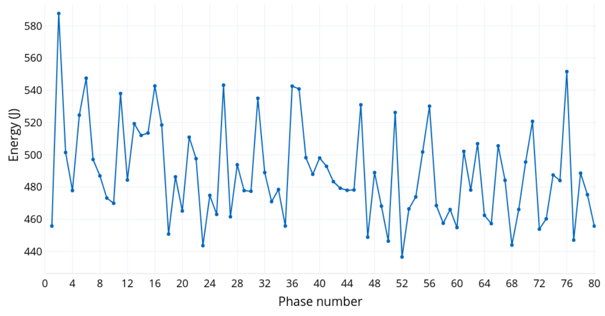
Rts	Clusters						
	1	2	3	4	5	6	7
/PhaseRegion/geofem_solver_11/ geofem_solver_cg_11	{1.6, 2.8}	{1.3, 2.0}	{1.4, 2.1}	{1.3, 3.0}	{1.3, 1.8}	{1.9, 2.0}	{1.3, 1.6}
/PhaseRegion/mat_ass_thermal/ mat_ass_thermal_main_361	{2.0, 1.4}	{2.5, 1.8}	{1.9, 1.2}	{2.5, 1.7}	{2.4, 1.3}	{2.3, 1.7}	{1.8, 2.0}

Theoretical Savings

Table 7.5 presents the theoretical energy savings in percentages computed by the *interphase* tuning plugin when the phases are executed with their corresponding cluster-best configurations, and the rts's with the rts-specific best configurations. The savings computed by the plugin are theoretical, in that they represent



(a) Single node run.



(b) Multi-node run.

Figure 7.5: The trend in the energy consumption across the phases of 128.GAPgeofem.

the theoretical maximum savings that could be obtained if the 80 phases that were seen during DTA are executed again with their best configurations. These values make no assumptions about the actual switching overhead that is incurred during production runs.

Column 1 presents the run type (single node or multi-node), column 2 presents the static savings for the phase, which arise from aggregating the improvement in the normalized energy across all the clusters when the phase is run with its cluster-best configuration. Static savings for the rts's (column 3) arise from aggregating the improvement in the normalized energy across all the clusters when the rts's are run with the static cluster-best configuration. Dynamic savings for the rts's (columns 4 and 5) arise from aggregating the improvement in the normalized energy across all the clusters when the rts's are run with the rts-specific best configurations from Table 7.3.

We observe high static savings of 38.39% for the single node run and 22.9% for multi-node run. Static savings for the rts's, however, are not that high, and account to 3.45% and 4.53% for the single-node and multi-node run respectively. We can also observe high dynamic savings for the rts's w.r.t. the cluster-best (static-best) configuration as well as the default system configuration.

Table 7.5: Energy savings (static savings for the phase and the rts's, and dynamic savings for the rts's w.r.t. static-best and default configurations respectively), computed using the *interphase* plugin for 128.GAPgeofem.

Run configuration	**Static savings for phase (%)**	**Static savings for rts's (%)**	**Dynamic savings for rts's w.r.t. static-best config. (%)**	**Dynamic savings for rts's w.r.t. default config. (%)**
Single node	38.39	3.45	16.23	19.12
Multi-node	22.90	4.53	12.69	16.65

7.3.1.2 Evaluation of Runtime Application Tuning

During production runs, the RRL reads the tuning model containing the best found configurations for the phase and the rts's. Figure 7.5 illustrates the trend in the energy consumption for the untuned version on a single node and multiple compute nodes for the 80 DTA phases. The X-axis represents the phase number, and the Y-axis represents the node energy consumed in Joules.

Table 7.6 shows the runtime savings obtained for the job energy (column 2), CPU energy (column 3) and the execution time (column 4) for 128.GAPgeofem for the three cluster predictors: Markov chain, one-bit and two-bit cluster predictors. As we can see, the three predictors result in job and CPU energy savings of over 5% for the single node run, with the two-bit cluster predictor resulting in the highest runtime savings. It also produced the least performance (time-to-solution) degradation of only 0.39%, as indicated by the red arrow.

We see that all three predictors perform much better for multi-node runs, with the two-bit predictor resulting in the highest job and CPU energy savings of 10.64% and 13.56% respectively, followed by the Markov chain predictor, which results in CPU energy savings of 11.18%. While the Markov chain predictor comes second to the two-bit cluster predictor in terms of the energy savings, it produces the least performance degradation of 3.73%, as indicated by the red arrow.

Table 7.6: Runtime savings obtained for 128.GAPgeofem over the untuned version using Markov chain, one-bit and two-bit runtime cluster predictors.

Run configuration	**Job energy (%)**			**CPU energy (%)**			**Execution time (%)**		
	Markov chain	**One bit**	**Two bit**	**Markov chain**	**One bit**	**Two bit**	**Markov chain**	**One bit**	**Two bit**
Single node	6.06 ↓	5.71 ↓	6.5 ↓	6.03 ↓	6.51 ↓	6.68 ↓	-1.37 ↑	-0.58 ↑	-0.39 ↑
Multi-node	8.34 ↓	5.63 ↓	10.64 ↓	11.18 ↓	8.74 ↓	13.56 ↓	-3.73 ↑	-9.7 ↑	-5.22 ↑

7.3.2 sam(oa)^2

sam(oa)^2 [127], which stands for Space-filling Curves and Adaptive Meshes for Oceanic And Other Applications and developed at the Technische Universität München performs the simulation of a tsunami wave propagation, and a two-phase porous media flow. It can also be used on all finite-element-type or finite-volume-type applications that are based on matrix-free, element-oriented formulations. The simulation of a tsunami wave propagation uses the Sierpinski space-filling curve traversal for its adaptive triangular meshes, and is performed by solving a system of time-dependent Partial Differential Equations (PDEs) [128], called shallow water equations (SWE). SWEs describe the behaviour of fluids of a certain depth over time, based on some initial condition with the assumption that the effect of flow in the vertical direction can be neglected, thus generating a two-dimensional domain.

sam(oa)^2 uses ASAGI (a pArallel Server for Adaptive GeoInformation) [129], an open-source library to read geographical data from NetCDF input files in the form of a Cartesian grid for parallel simulations with adaptive mesh refinement. ASAGI distributes the geographic datasets over all compute nodes only on-demand, so that only a portion of the dataset is stored on each node. The inputs for the simulation include the initial conditions, such as changes in the ocean floor caused by an earthquake, conditions of the uneven ocean floor, i.e., elevation of the sea floor and bathymetry data, as well as the height and velocity [130].

The simulation first sets an initial state and then refines the grid incrementally until the user-defined refinement level is reached. Next, it applies a displacement to the bathymetry data and the water height to create the initial tsunami wave, and finally, the time stepping phase executes the simulation on the generated grid, combined with adaptive mesh refinement in each iteration. The simulation domain, which is the ocean, is discretized with a Cartesian grid, and the unknowns of the PDE are placed at various grid elements (grid cells, vertices or edges), and are solved by computing the values at each grid element [131]. The computation of new cell states involves the exchange of boundary cell layers between neighboring processes, and the computation of fluxes between cells is done using Augmented Riemann solvers [128].

For our experiments, we used the hybrid MPI+OpenMP version of sam(oa)^2, and compiled it for MPI, while disabling OpenMP. The experiments used the 2D bathymetry input dataset from GEBCO [132] generated for the Tohoku tsunami in 2011 near the east coast of Japan. We also set the maximum refinement level to 20. Figure 7.6 illustrates the sequence of grid refinements performed for a refinement depth of 20 as the simulation progresses through the tsunami time steps executed using 24 MPI processes after initially performing the earthquake displacement time steps.

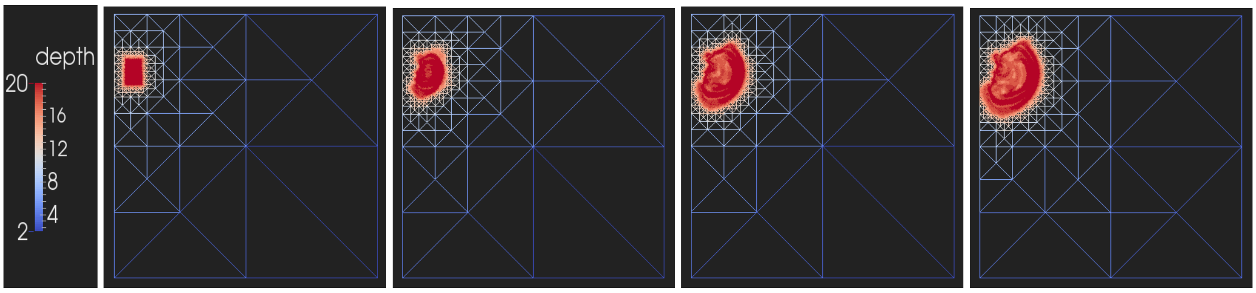


Figure 7.6: The grid refinement procedure in sam(oa)^2 as the simulation progresses through the tsunami time steps.

7.3.2.1 Evaluation of Design-Time Analysis

The phase region for sam(oa)^2 was determined using a combination of the domain knowledge from the application expert and inspection of the *cubex* file using CUBE. The application first performs small time steps that include a displacement to simulate the earthquake, and then performs the regular tsunami time steps after the earthquake is over. We annotated the second loop, i.e., the one that performs the tsunami time steps as the phase region.

Table 7.7 presents the result of the significant region analysis and dynamism detection by *readex-dyn-detect* for sam(oa)^2. The region *traverse_section* performs an Euler time stepping, and its rts's exhibit intra-phase dynamism w.r.t. variations in both compute intensity and the execution time, while the region *traverse_grids* performs an adaptive time stepping, and exhibits intra-phase dynamism due to the variations in only the compute intensity. This is because the size of the adaptive time step changes as the calculation proceeds in order to control the errors and ensure stability properties so that a constant simulation execution time is maintained.

Table 7.7: Significant regions identified by *readex-dyn-detect* for sam(oa)^2.

Significant region	**Rts call-path**	**Intra-phase dynamism**	
		Compute intensity	**Execution time**
swe_euler_timestep.traverse_section	/PhaseRegion/eulertraverse/ eulertraversewrapper/eulertraversesection	✔	✔
swe_adapt.traverse_grids	/PhaseRegion/traverseinplace/ traverseoutofplace/traversegrids	✔	✘

Single node

For single node experiments, we ran the application on a single compute node on Taurus using 24 MPI processes on 24 cores. We set the number of *samples*, i.e., the number iterations to run during a tuning step to 120. This means that we performed cluster analysis for these 120 points during DTA.

Cluster analysis

In Figure 7.7, the points on the graph represent the 10 eigenvalues for the 10 noise points obtained as a result of DBSCAN on sam(oa)^2. The X-axis represents the number of eigenvalues, and the Y-axis represents the eigenvalues for the normalized graph Laplacian matrix. We can see that there are small changes between the first two eigenvalues, and a large eigengap between the second and third eigenvalues, as shown by a red line. Thus, the value k, representing the input for the number of clusters for K-means at the end of spectral clustering was selected as 2.

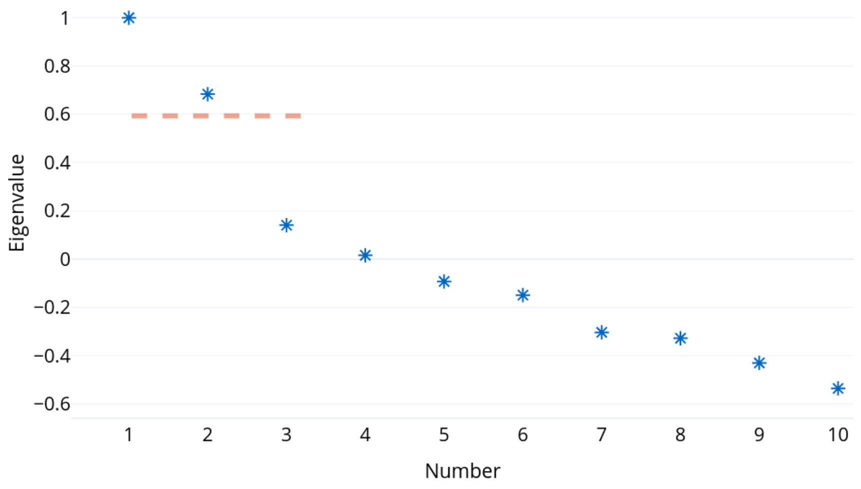


Figure 7.7: Eigenvalues computed for the graph Laplacian matrix for sam(oa)^2 for a single node run.

Figure 7.8 illustrates the final clustering consisting of four clusters obtained after performing DBSCAN and spectral clustering for sam(oa)^2. Clusters 1 and 2 result from DBSCAN, while clusters 3 and 4 result from spectral clustering. We can see that the majority of the points lie in cluster 1, since the points lie close to each other. The best configuration of the CPU and uncore frequencies for each cluster is defined as {CPU_freq, uncore_freq}. Cluster 2, which has a high value of normalized compute intensity has a high setting for the CPU frequency, and a low setting for the uncore frequency. Conversely, cluster 3 has a low setting for the CPU frequency since it is identified by low levels of compute intensity. For cluster 4, a combination of high CPU frequency and medium-high uncore frequency is optimal, since it lies in a region of medium levels of both compute intensity and L2 cache misses.

Best rts-specific configurations

Table 7.8 lists the rts-specific best configurations for the rts's of sam(oa)^2 for each cluster. The rts's are identified by their call-paths, and the best configurations are specified by the {CPU_freq, uncore_freq} settings. As we can see, the cluster-best configurations for the phases depicted in Figure 7.8 are not necessarily the best configurations for the rts's of the significant regions. The best configuration for the rts's of *swe_euler_timestep.traverse_section* for all the clusters has a high setting for the CPU frequency, and a low setting for the uncore frequency, except in cluster 4. Thus, we can say that this region is compute-bound. A similar behavior is seen for the rts's of the region *swe_adapt.traverse_grids* for clusters 1 and 2, while clusters 3 and 4 have a lower setting of the CPU frequency and a slightly higher value of the uncore frequency.

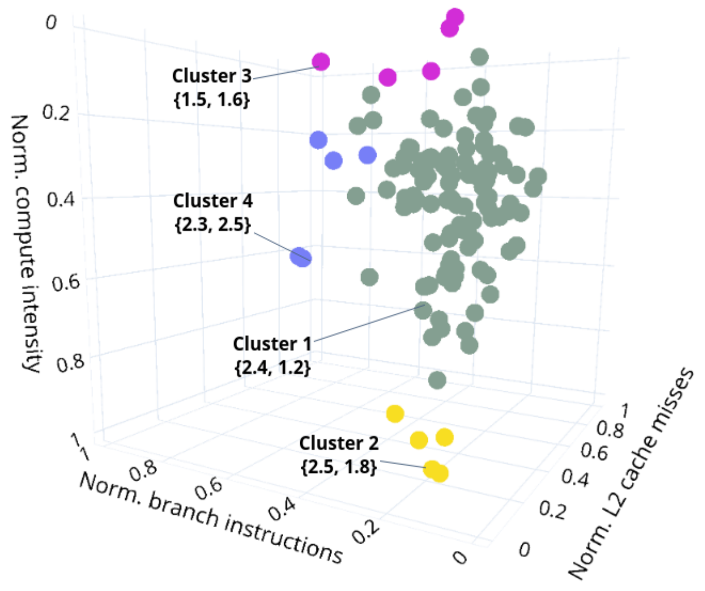


Figure 7.8: Single node results of the cluster analysis (DBSCAN followed by spectral clustering) performed on 120 phases of sam(oa)^2. Four clusters are produced, and the best configuration for each cluster is depicted in the form {CPU_freq, uncore_freq} .

Table 7.8: Rts-specific cluster-best configurations of the tuning parameters {CPU_freq, uncore_freq} for the rts's of the significant regions of sam(oa)^2 for a single node run.

Rts	**Clusters**			
	1	**2**	**3**	**4**
/PhaseRegion/eulertraverse/ eulertraversewrapper/eulertraversesection	{2.3, 1.8}	{2.5, 1.8}	{2.2, 1.3}	{2.3, 2.5}
/PhaseRegion/traverseinplace/ traverseoutofplace/traversegrids	{2.4, 1.2}	{2.5, 1.8}	{1.5, 1.6}	{1.9, 2.1}

Multiple nodes

For multi-node experiments, we ran the application on a two compute nodes on Taurus using 48 MPI processes on 48 cores. We set the number of *samples* to 110 to see the effect of changing the number of sample points on the clustering, and ultimately, the selection of best configurations.

Cluster analysis

In Figure 7.9, the points on the graph represent the 7 eigenvalues for the 7 noise points obtained as a result of DBSCAN on sam(oa)^2. The X-axis represents the number of eigenvalues, and the Y-axis represents the eigenvalues for the normalized graph Laplacian matrix. The large eigengap between the first and the second eigenvalues defines the value k as 1.

Figure 7.10 illustrates the final clustering consisting of three clusters obtained after performing DBSCAN and spectral clustering for sam(oa)^2. Clusters 1 and 2 result from DBSCAN, while cluster 3 results from spectral clustering. There are three noise points colored in red, which were not clustered by both DBSCAN

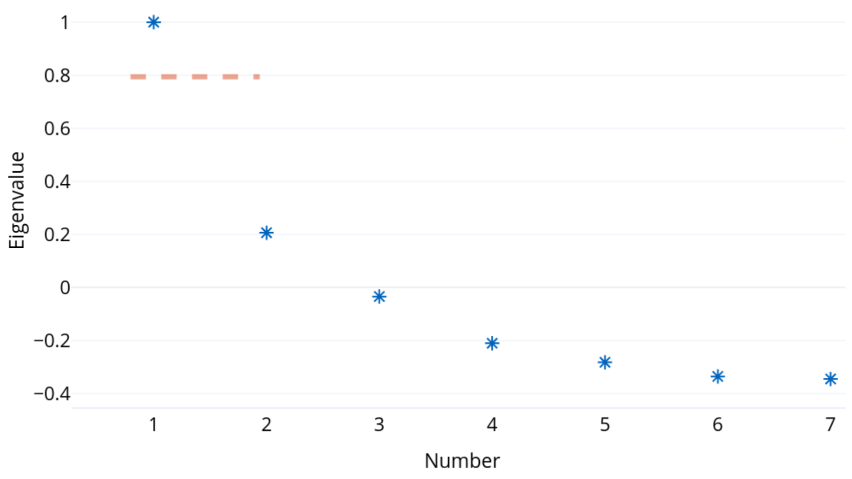


Figure 7.9: Eigenvalues computed for the graph Laplacian matrix for sam(oa)^2 for a multi-node run.

and spectral clustering. The clusters are separated by the number of normalized conditional branch instructions and L2 cache misses. The best configuration of the CPU and uncore frequencies for each cluster is illustrated by the {CPU_freq, uncore_freq} setting. The best configuration for cluster 1 is a high setting for the CPU frequency and a low uncore frequency resulting from the low number of L2 cache misses and conditional branch instructions. Clusters 2 and 3 have a medium-high setting for both core and uncore frequencies, corresponding to the rise in L2 cache misses and conditional branch instructions.

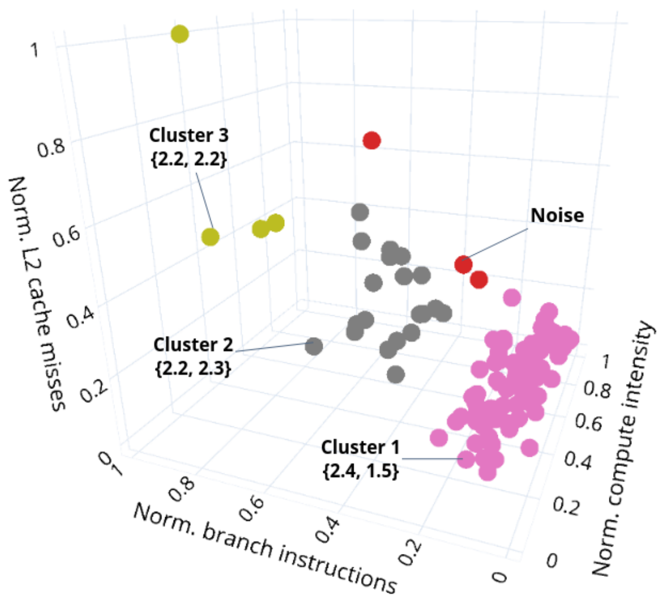


Figure 7.10: Multi-node results of the cluster analysis (DBSCAN followed by spectral clustering) performed on 110 phases of sam(oa)^2. Three clusters are produced, and the best configuration for each cluster is depicted in the form {CPU_freq, uncore_freq}.

Best rts-specific configurations

Table 7.9 lists the rts-specific best configurations for the rts's of sam(oa)^2 for each cluster. The rts's are identified by their call-paths, and the best configurations are specified by the {CPU_freq, uncore_freq} settings. The best configurations for the rts's follow a similar trend as the single node run, with a high setting for the CPU frequency. The adaptive traversal region, however shifts from being compute-bound to memory-bound in cluster 3, resulting in a higher setting for the uncore frequency.

Table 7.9: Rts-specific cluster-best configurations of the tuning parameters {CPU_freq, uncore_freq} for the rts's of the significant regions of sam(oa)^2 for a multi-node run.

Rts	**Clusters**		
	1	**2**	**3**
/PhaseRegion/eulertraverse/eulertraversewrapper/eulertraversesection	{2.5, 1.5}	{2.2, 2.3}	{2.4, 1.2}
/PhaseRegion/traverseinplace/traverseoutofplace/traversegrids	{2.5, 2.9}	{2.2, 2.3}	{1.6, 2.5}

Theoretical Savings

Table 7.10 presents the theoretical energy savings in percentages computed by the *interphase* tuning plugin for 120 phases for the single node run, and 110 phases for the multi-node run. Column 1 presents the run type (single node or multi-node), column 2 presents the static savings for the phase, column 3 represents the static savings for the rts's, and columns 4 and 5 represent the dynamic savings for the rts's.

We observe high static savings of 24% for the single node run and 33% for the multi-node run respectively. Static and dynamic savings for the rts's, however, are not that high for the single node run. On the other hand, the static and dynamic savings (w.r.t. the default configuration) show a dramatic increase for the rts's at 25.67% and 28.82% respectively for the multi-node run.

Table 7.10: Energy savings (static savings for the phase and the rts's, and dynamic savings for the rts's w.r.t. static-best and default configurations respectively), computed using the *interphase* plugin for sam(oa)^2.

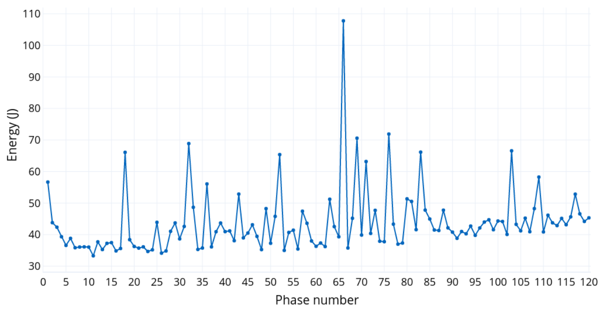
Run configuration	**Static savings for phase (%)**	**Static savings for rts's (%)**	**Dynamic savings for rts's w.r.t. static-best config. (%)**	**Dynamic savings for rts's w.r.t. default config. (%)**
Single node	24.06	5.3	4.17	9.25
Multi-node	33.16	25.67	4.24	28.82

7.3.2.2 Evaluation of Runtime Application Tuning

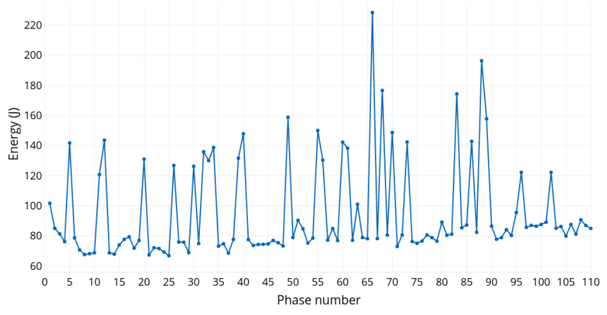
Figure 7.11 illustrates the trend in the energy consumption for the untuned version of sam(oa)^2 for the phases of DTA on a single node and multiple compute nodes. The X-axis represents the phase number, and the Y-axis represents the node energy consumed in Joules. We can see that the variations in the energy consumption are more drastic for the multi-node run as compared to the single-node run.

Table 7.11 shows the dynamic runtime savings obtained for the job energy (column 2), CPU energy (column 3) and the execution time (column 4) for sam(oa)^2 for the three cluster predictors: Markov chain, one-bit and two-bit cluster predictors.

As we can see, the three predictors result in significant savings for the job and CPU energy as well as the execution time in the range 16.9%-19.14% across single and multi-node runs. The improvement in the performance can also be attributed to switching to a higher uncore frequency during MPI communications.



(a) Single node run.



(b) Multi-node run.

Figure 7.11: The trend in the energy consumption across the phases of sam(oa)^2.

Table 7.11: Runtime savings obtained for sam(oa)^2 over the untuned version using Markov chain, one-bit and two-bit runtime cluster predictors.

Run configuration	**Job energy (%)**			**CPU energy (%)**			**Execution time (%)**		
	Markov chain	**One bit**	**Two bit**	**Markov chain**	**One bit**	**Two bit**	**Markov chain**	**One bit**	**Two bit**
Single node	17 ↓	16.9 ↓	17.1 ↓	17.32 ↓	17.07 ↓	17.25 ↓	17.2 ↓	17.4 ↓	17.73 ↓
Multi-node	18.31 ↓	18.14 ↓	18.61 ↓	18.4 ↓	18.15 ↓	18.6 ↓	18.85 ↓	18.81 ↓	19.14 ↓

The two-bit cluster predictor produced the highest savings in all three parameters for both single node and multi-node experiments. The Markov chain predictor follows closely, and even gaining slightly more savings of 17.32% for the CPU energy for the single node run.

7.3.3 INDEED

INDEED [3] is a finite element software with an implicit time integration, and is developed by GNS (Gesellschaft für Numerische Simulation mbH). INDEED offers high-precision calculation models for the simulation of forming processes, such as modeling of large plastic deformations, and determining stress distribution during sheet metal forming processes.

The simulation involves a stationary workpiece and a number of tools with different geometries that move towards this workpiece [133]. When there is a contact between the tool and the workpiece, the software performs an adaptive mesh refinement, where the number of finite element nodes that it uses increases between time steps, which means that the computational cost also increases with the rising number of elements. When the tools come in contact with the workpiece for the first time, a lot of computational work is done to deal with this by refining the mesh. In order to keep the problem tractable, INDEED then reduces the time step, thereby reducing the computational cost.

The simulation code in INDEED has more than 2000 subroutines, of which less than half will actually be used in any given run. The invocation of a specific routine depends on the design of the input data set, which involves selecting various mathematical and engineering properties. These include different types of elements (membranes, shells, volume elements), types of contact (rigid/deformable and deformable/deformable), types of material (mild steel, high strength steel, aluminum, . . .), friction models, types of tool control (path control, force control, rotation, bending, . . .), types of operations (punching of holes, trimming, . . .), and surface and volume load among other properties.

The simulation in INDEED consists of three loops: time loop, contact loop and equilibrium loop. At the start of the time loop, which is the outermost loop, the algorithm checks if there is a new tool, configures the initial settings and performs an iteration procedure for each time step in order to produce an equilibrium of simulated forces. Then, it starts the contact loop upon the contact of the tool with the workpiece. Each iteration of the contact loop executes the innermost loop, which is the equilibrium loop, which solves a system of equations until equilibrium is achieved. The contact loop is executed until there is no more change in the contact points. Depending on what happens mechanically in the simulated process at the current time step, it may take a larger or a smaller number of iterations to reach equilibrium. As a result of the implicit time integration, the computational cost is relatively high. On the other hand, it provides a high degree of accuracy of the numerical solutions.

INDEED offers an MPI and an OpenMP version, but our work is concentrated on the OpenMP version because it is more important from the perspective of the users.

7.3.3.1 Evaluation of Design-Time Analysis

We used 12 OpenMP threads to run INDEED with the input data set for a steel sheet on a single compute node of Taurus. The steel benchmark runs for a total of 154 iterations of the time loop, and results in a long execution time per phase. We turned off Score-P instrumentation for the threads because of the bug in the Score-P OA interface, as mentioned earlier.

Table 7.12 presents the result of the significant region analysis and dynamism detection by *readex-dyn-detect*. As we can see, five significant regions were detected for INDEED. The region *restou* has two rts's that can be identified by their different call-paths. However, they exhibit intra-phase dynamism due to variation in only the compute intensity. Of the other significant regions, only *write_var* and *beltop* show variations in the execution time.

Table 7.12: Significant regions identified by *readex-dyn-detect* for INDEED.

Significant region	**Rts call-path**	**Intra-phase dynamism**	
		Compute intensity	**Execution time**
write_var	/PhaseRegion/cps04/write_eval_data/write_var	✔	✔
restou	/PhaseRegion/restou	✔	✘
	/PhaseRegion/rezvor/wrdump/restou	✔	✘
lines3	/PhaseRegion/cps02/itsteu/lines3	✔	✘
beltop	/PhaseRegion/vsorel1/vsorelf/beltop	✔	✔
crit04	/PhaseRegion/cps01/geocr4/crit04	✔	✘

The annotation of the phase region in INDEED is slightly more complicated than the other applications since there are three loops, i.e., time loop, contact loop and equilibrium loop that have varying levels of dynamism. The user may annotate any of these loops as the phase region. We used a combination of application expert knowledge and visualization of the *cubex* file using CUBE to identify the phase region. We observed that the time loop, i.e., the outermost loop and the equilibrium loop exhibited the most inter-phase dynamism. Although annotating the equilibrium loop as the phase region is a valid choice, we used the time loop as the phase region for our experiments due to its longer running time per phase.

To perform DTA, we set the number of *samples*, i.e., the number experiments or iterations of the simulation loop to 110. This means that we performed cluster analysis for these 110 points during DTA.

Cluster analysis

In Figure 7.12, the points on the graph represent the 12 eigenvalues for the 12 noise points obtained as a result of DBSCAN on INDEED. The X-axis represents the number of eigenvalues, and the Y-axis represents the eigenvalues for the normalized graph Laplacian matrix. We can see that there is a large eigengap between the second and third eigenvalues, as shown by a red line. Thus, the value of *k* was selected as 2.

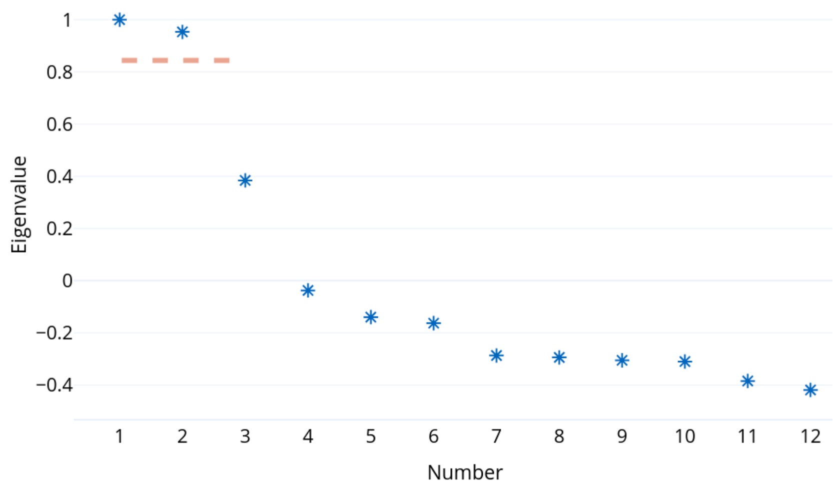


Figure 7.12: Eigenvalues computed for the graph Laplacian matrix for INDEED for a single node run.

Figure 7.13 illustrates the final clustering consisting of five clusters obtained after performing DBSCAN and spectral clustering for INDEED. Clusters 1-3 result from DBSCAN, while clusters 4 and 5 result from spectral clustering. The clusters have a marked distinction between them due to the variation in the normalized L2 cache misses. The exception is cluster 5, which has a much lower range of the L2 cache misses, and a higher degree of variation in the compute intensity.

The best configuration of the CPU and uncore frequencies for each cluster is illustrated by the {CPU_freq, uncore_freq} setting. We see that for clusters that lie in a region of low normalized L2 cache misses, a lower setting of the uncore frequency is recommended. A higher amount of traffic entering the L2 cache results in a higher setting of the uncore frequency, as indicated by clusters 1, 2 and 4. This means that a higher uncore frequency is beneficial when the data is outside the L2 cache because accesses can happen much faster. It is also clear that INDEED benefits from a high setting for the CPU frequency due to the adaptive mesh refinement.

Best rts-specific configurations

Table 7.13 lists the rts-specific best configurations for the rts's of INDEED for each cluster. The rts's are identified by their call-paths, and the best configurations are specified by the {CPU_freq, uncore_freq} settings. We can see that the rts's of *restou* have the same optimal configurations across all the clusters. This means that the intra-phase variations in the compute intensity for this region had no effect on the selection of the optimal configuration. The rts's of the region *write_var* are not called in the phases of clusters 4 and 5, so no optimal configuration exists for these clusters. It can also be observed that the best configuration for the cluster is not necessarily the best configuration for the rts's, and that all the rts's with the exception of */PhaseRegion/cps02/itsteu/lines3* have a medium-high setting for the CPU frequency in the range of 1.9-2.5 GHz for all the clusters.

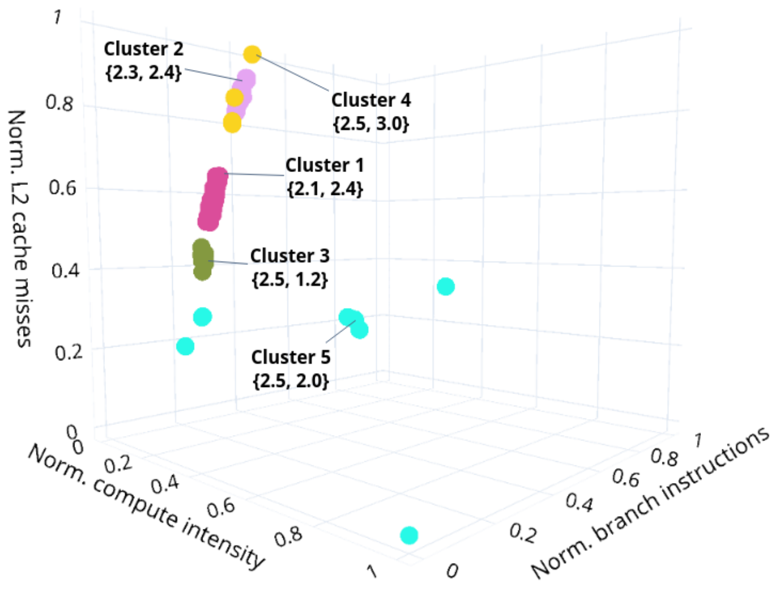


Figure 7.13: Single node results of the cluster analysis (DBSCAN followed by spectral clustering) performed on 110 phases of INDEED. Five clusters are produced, and the best configuration for each cluster is depicted in the form {CPU_freq, uncore_freq}.

Table 7.13: Rts-specific cluster-best configurations of the tuning parameters {CPU_freq, uncore_freq} for the rts's of the significant regions of INDEED for a single node run.

Rts	**Clusters**				
	1	**2**	**3**	**4**	**5**
/PhaseRegion/cps04/write_eval_data/write_var	{2.5, 2.6}	{2.3, 2.4}	{1.9, 1.4}	-	-
/PhaseRegion/restou	{2.5, 1.5}	{2.4, 1.8}	{2.5, 1.2}	{1.9, 2.2}	{2.5, 2.0}
/PhaseRegion/rezvor/wrdump/restou	{2.5, 1.5}	{2.4, 1.8}	{2.5, 1.2}	{1.9, 2.2}	{2.5, 2.0}
/PhaseRegion/cps02/itsteu/lines3	{1.7, 2.5}	{2.1, 2.4}	{2.2, 1.9}	{1.9, 1.2}	{2.2, 1.9}
/PhaseRegion/vsorel1/vsorelf/beltop	{2.5, 1.5}	{2.3, 1.3}	{2.5, 1.2}	{2.2, 2.2}	{2.5, 2.0}
/PhaseRegion/cps01/geocr4/crit04	{2.1, 2.8}	{2.3, 2.4}	{2.3, 2.0}	{1.9, 1.2}	{2.5, 2.0}

Theoretical Savings

Table 7.14 presents the theoretical energy savings in percentages computed by the *interphase* tuning plugin for the phase and rts's of INDEED. Column 1 presents the run type (single node), column 2 presents the static savings for the phase, column 3 represents the static savings for the rts's, and columns 4 and 5 present the dynamic savings for the rts's.

We observe maximum savings of 15.02% for the phase. The static savings for the rts's amount to 7.16%, while the dynamic savings w.r.t. the static-best configuration is 4.06%, and default configuration is 10.93%. Since all the clusters of the phases of INDEED benefit from a high setting for the CPU frequency, we can infer that the static savings for the phase are majorly influenced by the uncore frequency switching. Thus, high CPU frequency values combined with uncore frequency switching can theoretically produce good savings for the phases and the rts's.

Table 7.14: Energy savings (static savings for the phase and the rts's, and dynamic savings for the rts's w.r.t. static-best and default configurations respectively), computed using the *interphase* plugin for INDEED.

Run configuration	**Static savings for phase (%)**	**Static savings for rts's (%)**	**Dynamic savings for rts's w.r.t. static-best config. (%)**	**Dynamic savings for rts's w.r.t. default config. (%)**
Single node	15.02	7.16	4.06	10.93

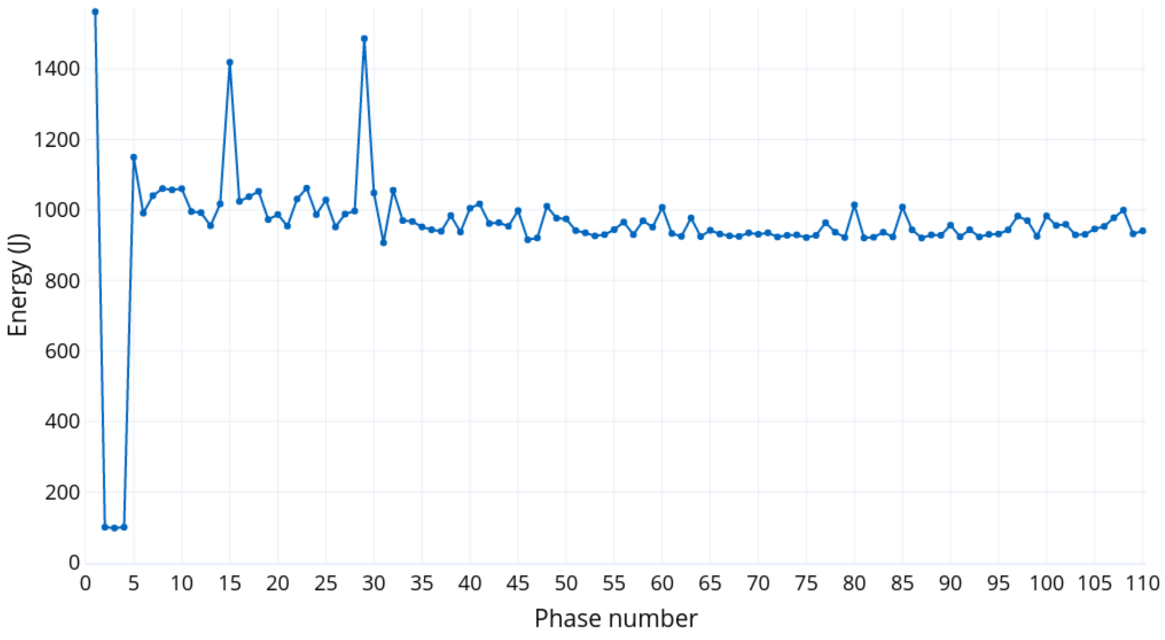


Figure 7.14: The trend in the energy consumption across the phases of INDEED.

7.3.3.2 Evaluation of Runtime Application Tuning

During production runs, the RRL reads the tuning model containing the best found configurations for the phase and the rts's. Figure 7.14 illustrates the trend in the energy consumption for the 110 phases of DTA for the untuned version on a single compute node of Taurus. The X-axis represents the phase number, and the Y-axis represents the node energy consumed in Joules. As we can see, the variations in the energy consumption across the initial 30-40 phases are drastic as a result of the adaptive mesh refinement.

Table 7.15 shows the dynamic runtime savings obtained for the job energy (column 2), CPU energy (column 3) and the execution time (column 4) for INDEED for the Markov chain, one-bit and two-bit cluster predictors. As we can see, the Markov chain predictor outperforms the one-bit and the two-bit cluster predictors in all the three parameters: job energy with 6.4% savings, CPU energy with 6.4% savings, and time-to-solution with 7.7% savings. The one-bit predictor also seems to do very well for the three aspects. Moreover, all three predictors improve the time-to-solution for the single node run.

From the above plots and figures, it is evident that it is always advisable to choose the highest possible CPU frequency. The optimal choice of the other tuning parameters, such as the uncore frequency and the number of OpenMP threads depends on the objective function. For example, varying the number of OpenMP threads results in a lower runtime, but does not have much impact on the energy savings. Since we were unable to optimize this tuning parameter, we cannot make further inferences.

Table 7.15: Runtime savings obtained for INDEED over the untuned version using Markov chain, one-bit and two-bit runtime cluster predictors.

Run configuration	**Job energy (%)**			**CPU energy (%)**			**Execution time (%)**		
	Markov chain	**One bit**	**Two bit**	**Markov chain**	**One bit**	**Two bit**	**Markov chain**	**One bit**	**Two bit**
Single node	6.4↓	6.1↓	5.44↓	6.4↓	6.04↓	5.54↓	7.7↓	7.3↓	7.04↓

7.3.4 miniMD

miniMD is a lightweight, parallel Molecular Dynamics (MD) micro-benchmark in the Mantevo benchmark suite maintained by Sandia National Laboratories. The code is written in C++, and performs parallel molecular dynamics simulation of a Lennard-Jones or an Embedded Atom Model (EAM) system, and is a weakly scaling benchmark [134]. The benchmark simulates the behavior of atoms, and describes the interaction between two uncharged molecules or atoms under the influence of different forces. miniMD uses spatial decomposition MD, where the 3D simulation space consisting of atoms is divided into cells of dimensions equal to the sum of the cutoff distance and a margin, and subsets of cells are owned by individual processors [135].

In each iteration of the time loop, short-range force calculations are performed for all pairs of atoms that are within the cutoff distance using a pre-computed list of near neighbors for each atom. This is done to avoid evaluating the distance between all pairs of atoms. The two most time consuming portions of the code are the force computation and the creation of the neighbor list for each atom. New forces are then calculated by considering the current neighbor list for each atom. Finally, inter-node communication is performed between the nodes to exchange the position and force information for the atoms near the boundaries.

During the simulation, the atoms may move from one node to another in a process known as atom exchange. The force acting on one atom may depend on another atom existing on another node, thus resulting in the need to cross node boundaries. miniMD uses MPI calls to perform the atom exchange by sending the positions of the dependent atoms from each node to its neighboring nodes before the force computation, and receiving some force contribution from the neighboring nodes after the force computation [136].

The benchmark takes in an input file, which enables users to specify the type of potential (Lennard-Jones or EAM), problem size, number of time steps, size of each time step, temperature, atom density, and particle interaction cutoff distance. We used the Lennard-Jones potential and set the reneighboring of atoms to be performed once every 10 iterations for a total of 100 time steps (or phases) for both single and multi-node experiments for miniMD.

7.3.4.1 Evaluation of Design-Time Analysis

Table 7.16 presents the result of the significant region analysis and dynamism detection by *readex-dyn-detect* tool. To instrument miniMD, we first identified the for-loop in the Integrate::run() function in integrate.cpp as the phase region. The for-loop takes the number of time steps from the input file, and calls three functions, namely *borders()*, *build()* and *compute()* that *readex-dyn-detect* returns as significant regions. As we can see, the region *borders()* has no dynamism w.r.t. either compute intensity or execution time. However, *readex-dyn-detect* returns it as a significant region since the granularity of this region is greater than the minimum threshold of 100 ms. Another interesting observation is that none of the regions have variations in the execution time.

Table 7.16: Significant regions identified by *readex-dyn-detect* for miniMD.

Significant region	Rts call-path	Intra-phase dynamism	
		Compute intensity	Execution time
build	/PhaseRegion/NEIGHBOR_BUILD	✔	✘
compute	/PhaseRegion/FORCELJ_COMP_HALF	✔	✘
borders	/PhaseRegion/COMM_BORDERS	✘	✘

We also observe that once every 10 phases, regions *build()* and *borders()* are called due to the reneighboring process. Consequently, during these phases, the *compute()* region contributes little to the energy consumption and the execution time. This is the source of inter-phase dynamism in miniMD. This periodicity in the dynamism is associated to one of the input parameters for miniMD - reneighboring atoms once every *N* steps/iterations.

To perform inter-phase analysis, we set the number of *samples* to 80 for both single node and multi-node experiments. This means that we performed cluster analysis for these 80 points during DTA.

Single node

For single node experiments, we ran the application on a single compute node on Taurus using 4 MPI processes on 24 cores. We configured miniMD with a problem size of 90x90x90 to simulate 2,916,000 atoms with a uniform density of 0.8442.

Cluster analysis

In Figure 7.15, the points on the graph represent the 10 eigenvalues for the 10 noise points obtained as a result of DBSCAN on miniMD. The X-axis represents the number of eigenvalues, and the Y-axis represents the eigenvalues for the normalized graph Laplacian matrix.

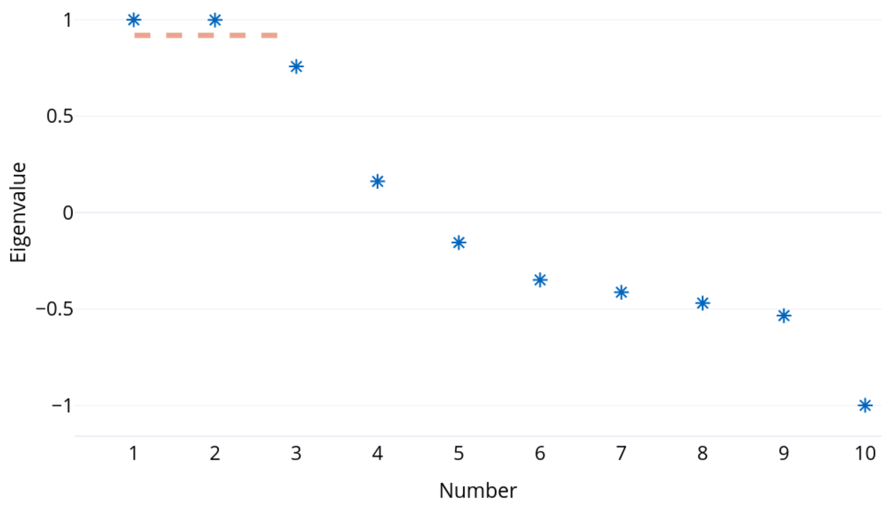


Figure 7.15: Eigenvalues computed for the graph Laplacian matrix for miniMD for a single node run.

We can see that there is one large eigengap between points 2 and 3 as shown by a red line, with essentially no gap between the first two points. Thus, the value *k*, representing the input for the number of clusters

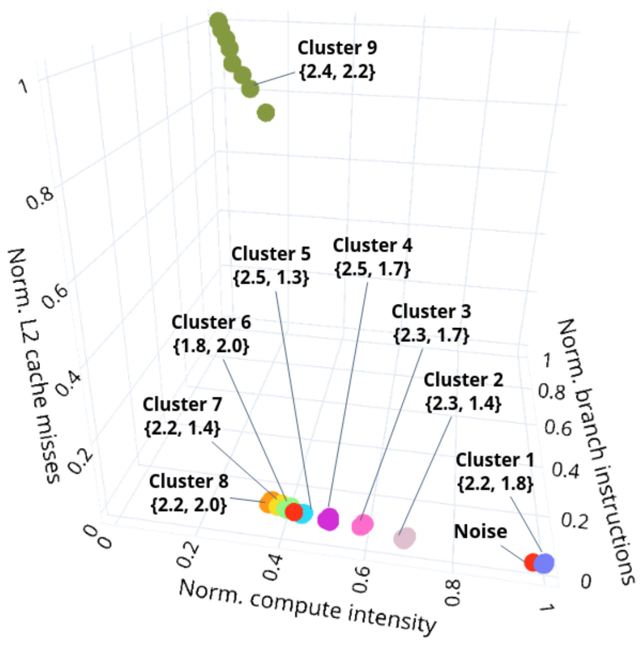


Figure 7.16: Single node results of the cluster analysis (DBSCAN followed by spectral clustering) performed on 80 phases of miniMD. Nine clusters are produced, and the best configuration for each cluster is depicted in the form {CPU_freq, uncore_freq}.

for K-means at the end of spectral clustering was selected as 2. After spectral clustering, it was observed that one of the clusters had fewer points than the minimum number of points, i.e., 4. Thus, this cluster was rejected by the plugin, and the points were marked as noise.

Figure 7.16 illustrates the final clustering consisting of nine clusters obtained after performing DBSCAN and spectral clustering for miniMD. Clusters 1-8 result from DBSCAN, while cluster 9 results from spectral clustering. We can see that the individual clusters are marked by varying levels of the normalized compute intensity, with cluster 1 having the highest values.

The best configuration of the CPU and uncore frequencies for each cluster is illustrated by the {CPU_freq, uncore_freq} setting. The best configurations for all the clusters have a medium-high setting for the CPU frequency, and a transition from low to higher values of the uncore frequency as we move from high compute intensity regions to low compute intensity regions. Cluster 9, which lies in an area of high L2 cache misses has the highest setting of the uncore frequency. Clusters 1-8 generally benefit from a high CPU frequency setting and a low uncore frequency setting.

Best rts-specific configurations

Table 7.17 lists the rts-specific best configurations for the rts's of miniMD for each cluster. The rts's are identified by their call-paths, and the best configurations are specified by the {CPU_freq, uncore_freq} settings. As we can see, only the region *compute()* is called in all the phases, and has a high setting for the core frequency and low setting for the uncore frequency (except in cluster 6). On the other hand, the regions *build()* and *borders()* are called only in cluster 9, since this cluster contains every tenth phase of the simulation loop. The core frequency setting for both regions is high, while the uncore frequency is understandably low for *build()* since it only builds the neighbor list. The high setting for the uncore frequency for *borders()* is due to the inter-node communication that is performed between the nodes to exchange the position and force information. This could indicate that it is more memory-bound.

Table 7.17: Rts-specific cluster-best configurations of the tuning parameters {CPU_freq, uncore_freq} for the rts's of the significant regions of miniMD for a single node run.

Cluster	**Rts's**		
	/PhaseRegion/ FORCELJ_COMP_HALF	**/PhaseRegion/ BUILD**	**/PhaseRegion/ COMM_BORDERS**
1	{2.1, 1.5}	-	-
2	{2.3, 1.4}	-	-
3	{2.3, 1.7}	-	-
4	{2.5, 1.7}	-	-
5	{2.5, 1.3}	-	-
6	{1.9, 2.9}	-	-
7	{2.2, 1.4}	-	-
8	{2.2, 2.0}	-	-
9	{2.4, 2.2}	{2.2, 1.4}	{2.3, 2.4}

Multiple nodes

For multi-node experiments, we ran the application on a two compute nodes on Taurus using 48 MPI processes on 48 cores. We configured miniMD with a problem size of 160x160x160 to simulate 16,384,000 atoms with a uniform density of 0.8442.

Cluster analysis

In Figure 7.17, the points on the graph represent the 12 eigenvalues for the 12 noise points obtained as a result of DBSCAN on miniMD. The X-axis represents the number of eigenvalues, and the Y-axis represents the eigenvalues for the normalized graph Laplacian matrix. We can see that there is a large eigengap between the second and third eigenvalues, as shown by a red line. Thus, the value k was selected as 2.

Figure 7.18 illustrates the final clustering consisting of seven clusters obtained after performing DBSCAN and spectral clustering for miniMD. Clusters 1-5 result from DBSCAN, while clusters 6 and 7 result from spectral clustering. Like in the case of the single node run, the clusters have varying degrees of normalized compute intensity. In addition, they also now display different levels of conditional branch instructions. We also observe that cluster 5 is larger than the other clusters since more points lie in close proximity to each other. There is, however, an outlier cluster, namely cluster 7, which consists of points lying close to clusters 5 and 1. This is due to the fact that these points were marked as noise by DBSCAN, and eventually clustered by spectral clustering into a single cluster because they were more similar to each other than the points from cluster 6.

The best configuration of the CPU and uncore frequencies for each cluster is illustrated by the {CPU_freq, uncore_freq} setting. As with the case of the single node run, it can be observed that all the clusters have a relatively high value for the best CPU frequency setting. The best configurations for clusters 1-5 and 7 are similar to the single node result, with a medium-high setting for the CPU frequency and a low-medium setting for the uncore frequency. We also observed that varying the problem size for the multi-node configuration produced similar results.

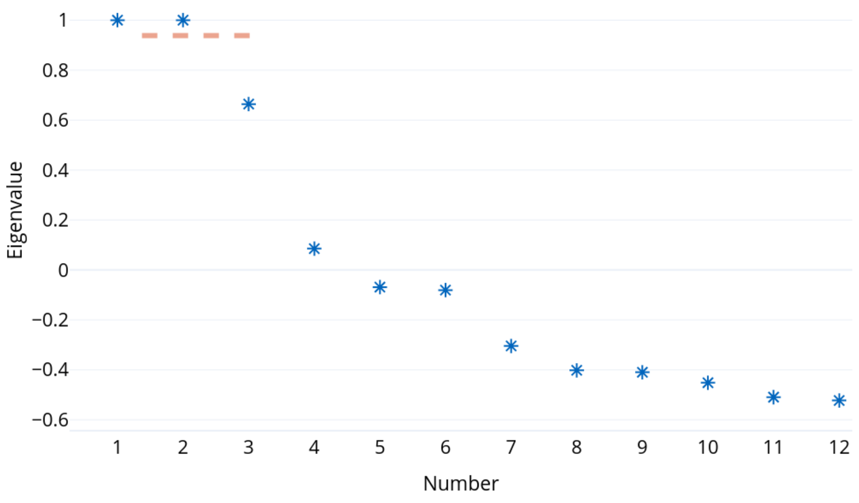


Figure 7.17: Eigenvalues computed for the graph Laplacian matrix for miniMD for a multi-node run.

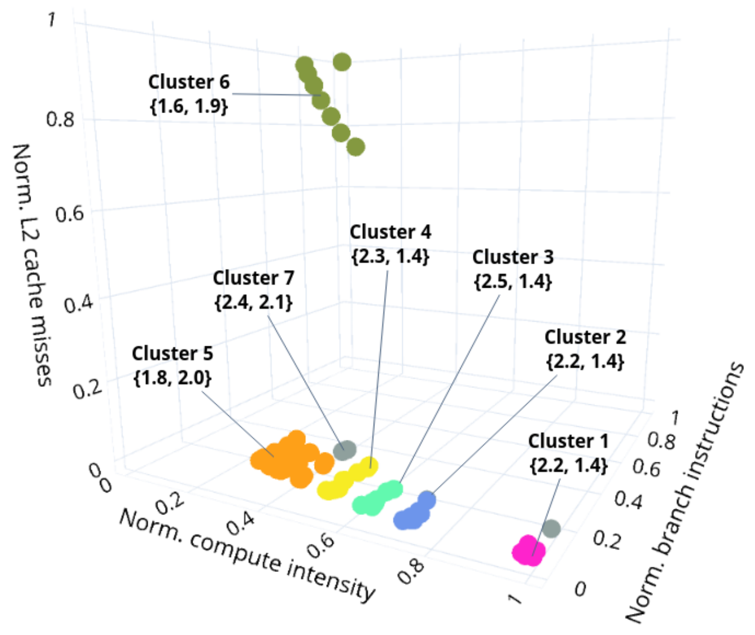


Figure 7.18: Multi-node results of the cluster analysis (DBSCAN followed by spectral clustering) performed on 80 phases of miniMD. Seven clusters are produced, and the best configuration for each cluster is depicted in the form {CPU_freq, uncore_freq}.

Best rts-specific configurations

Table 7.18 lists the rts-specific best configurations for the rts's of miniMD for each cluster. The rts's are identified by their call-paths, and the best configurations are specified by the {CPU_freq, uncore_freq} settings. The rts-specific best configurations for the region *compute()* for all the clusters except cluster 6 have a medium-high setting for the core frequency, and a low setting for the uncore frequency. For cluster 6, the best configuration is a low setting for the CPU frequency and a low-medium setting for the uncore frequency. This is because the phases of this cluster perform reneighboring by calling the functions *build()* and *borders()*. The best configuration for *borders()* is marked by a high setting for the uncore frequency, similar to the single node run.

Table 7.18: Rts-specific cluster-best configurations of the tuning parameters {CPU_freq, uncore_freq} for the rts's of the significant regions of miniMD for a multi-node run.

Cluster	Rts's		
	/PhaseRegion/ FORCELJ_COMP_HALF	**/PhaseRegion/ BUILD**	**/PhaseRegion/ COMM_BORDERS**
1	{2.2, 1.4}	-	-
2	{1.9, 2.0}	-	-
3	{2.5, 1.4}	-	-
4	{2.3, 1.4}	-	-
5	{2.2, 1.7}	-	-
6	{1.6, 1.9}	{1.9, 1.6}	{1.3, 2.9}
7	{2.4, 2.1}	-	-

Theoretical Savings

Table 7.19 presents the theoretical energy savings in percentages computed by the *interphase* tuning plugin for the phase and the rts's of miniMD. Column 1 presents the run type (single node or multi-node), column 2 presents the static savings for the phase, column 3 represents the static savings for the rts's, and columns 4 and 5 present the dynamic savings for the rts's.

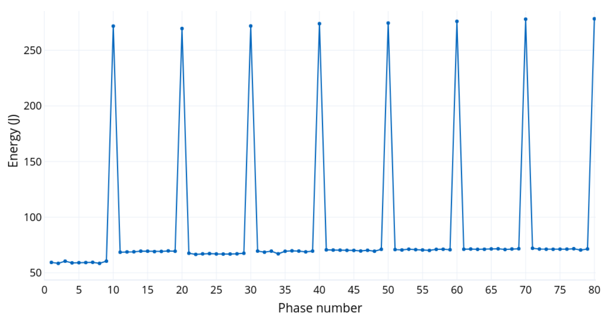
We observe maximum savings for the phase at 10.19% for the multi-node run. As expected, the dynamic savings for the rts's w.r.t. the static cluster-best configuration amounts to a maximum of 2.53% for the multi-node run, while the single node run shows very little savings. This is because of the execution of the function *compute()* for most of the application run, which does not give rise to much intra-phase dynamism. Thus, the rts's could potentially be run with the cluster-best configurations without much effect on the normalized energy consumption.

Table 7.19: Energy savings (static savings for the phase and the rts's, and dynamic savings for the rts's w.r.t static-best and default configurations respectively), computed using the *interphase* plugin for miniMD.

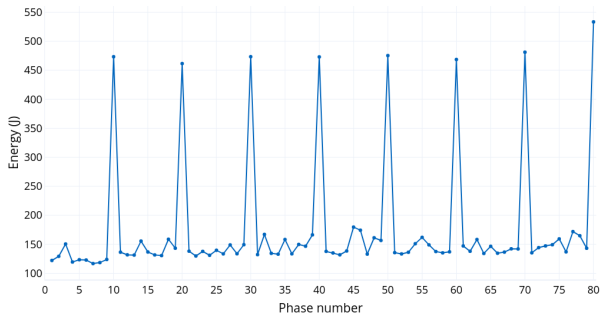
Run configuration	**Static savings for phase** (%)	**Static savings for rts's** (%)	**Dynamic savings for rts's w.r.t. static-best config.** (%)	**Dynamic savings for rts's w.r.t. default config.** (%)
Single node	7.96	7.8	0.14	7.93
Multi-node	10.19	5.95	2.53	8.34

7.3.4.2 Evaluation of Runtime Application Tuning

During production runs, the RRL reads the tuning model containing the best found configurations producing the lowest normalized energy consumption for the phase and the rts's. Figure 7.19 illustrates the trend in the energy consumption for the 80 phases of DTA for the untuned version on a single node and multiple compute



(a) Single node run.



(b) Multi-node run.

Figure 7.19: The trend in the energy consumption across the phases of miniMD.

nodes. The X-axis represents the phase number, and the Y-axis represents the node energy consumed in Joules. As we can see, the trend in the energy consumption for both runs show a similar effect where the tenth phase has a sharp increase in the energy as an effect of the reneighboring procedure. We can also see that the energy consumption for the multi-node run shows more variations between the phases as compared to the single node run.

Table 7.20 shows the runtime savings obtained for the job energy (column 2), CPU energy (column 3) and the execution time (column 4) for miniMD for the Markov chain, one-bit and two-bit cluster predictors. As we can see, the three predictors perform poorly in improving the CPU energy for the single-node run. The two-bit predictor outperforms the Markov chain and one-bit predictor for all the test cases for the single node run. Moreover, all predictors improve the time-to-solution for the single node run, with the two-bit predictor reducing the execution time by 12.7%. For multi-node runs, all three predictors perform nearly

Table 7.20: Runtime savings obtained for miniMD over the untuned version using Markov chain, one-bit and two-bit runtime cluster predictors.

Run configuration	**Job energy (%)**			**CPU energy (%)**			**Execution time (%)**		
	Markov chain	**One bit**	**Two bit**	**Markov chain**	**One bit**	**Two bit**	**Markov chain**	**One bit**	**Two bit**
Single node	3.0 ↓	3.45 ↓	6.6 ↓	0.21 ↓	0.33 ↓	0.8 ↓	4.2 ↓	4.2 ↓	12.7 ↓
Multi-node	4.39 ↓	4.46 ↓	4.41 ↓	8.03 ↓	8.08 ↓	8.34 ↓	-11.69 ↑	-11.69 ↑	-11.04 ↑

the same in reducing the job energy, CPU energy and the execution time, and result in much higher CPU energy savings than the single node run. On the contrary to the single node results, the predictors degrade the performance for multi-nodes by increasing the time-to-solution by nearly 11.7%.

The reason for low job energy savings for both single node and multi-node runs is because the cluster ids change every 10 phases, meaning that after every 10 phases, a new cluster is formed. Hence, during RAT, an unseen phase will always belong to a new cluster that was not identified during DTA. Thus, the predictors will always mark the unseen phase as noise. Hence, we can conclude that the cluster predictors perform poorly for applications like miniMD, where a new cluster is formed after every *n* phases.

8 Summary and Outlook

Energy-efficiency optimization still remains one of the major hurdles in reaching Exascale levels of computing. Current systems require more than 50 MW of power to reach the Exascale level, which is more than the power budget allocated for HPC systems. One of the biggest challenges in this area is the focus of application programmers in improving the performance of HPC applications while neglecting possible improvements in the energy/power consumption. Moreover, the lack of platform knowledge among developers even as hardware vendors have implemented multiple power saving features in modern processors, such as clock gating, power gating, clock modulation, DVFS and UFS, proves to be another challenge.

A key point is that pure hardware-based solutions cannot address the dynamism that applications exhibit. Thus, a software tuning approach that can free the users from learning the intrinsics of the application behaviour can be implemented with the help of autotuning. Autotuning enables to automatically walk a search space of tuning knobs, or the so-called tuning parameters using a search strategy to optimize a specific tuning objective. So far, existing tools generally rely on static tuning, where a single frequency is set for the entire application run. However, this approach has the drawback of being too coarse-grained by aiming for a "one solution fits all" approach, since most HPC applications typically exhibit changing characteristics. These changes, known as application dynamism in different regions of the program, cause the program execution to jump between compute-, memory-, and I/O-bound code regions. Thus, a more fine-grained approach where a different setting of the tuning parameters is applied to each program region to exploit the dynamic behavior of the application overcomes the static tuning approach.

To automate the tuning process, we presented a tools-aided approach by combining several pre-existing tools with novel runtime tuning tools to guide the optimal tuning of the hardware (core frequency and uncore frequency), the system software (number of OpenMP threads), and the application-level tuning parameters. As a foundation for our work, we introduced the formalism, including the definitions of the terms used in the rest of the thesis, and the tools (PTF, Score-P) and APIs (PAPI) that form the basis for our work. The work done in this thesis is an extension of the EU-funded Horizon2020 project READEX, which combined technologies from two ends of the spectrum by applying the system scenario methodology from the embedded systems domain to the HPC world to automate the process of determining the best settings of the tuning parameters, called system configurations.

We identified that the definition of a *phase* varies across previous works. Some define it as a period of execution with a stable behavior, which may contain multiple regions and iterations of the time loop. We define a phase as one iteration of the simulation loop that calls different program regions within a single run. Our work is also different from other related work, as we not only determine optimal configurations for different code regions when they exhibit varying behaviour (intra-phase dynamism), but even determine different best configurations for different executions of the same region, called runtime situations (rts's) depending on the execution context, e.g., the relaxation operation on a certain grid level. Beyond that, we also identify the data dependent variation in the characteristics of the phase over time, e.g., due to grid refinement, and compute best configurations for groups of similarly behaving phases. Such dynamic changes between the individual phases are called inter-phase dynamism.

Our work is based on a two-stage approach consisting of Design-Time Analysis (DTA), in which fine-granular application regions are first filtered out using *scorep-autofilter* to reduce the overhead of instrumentation. Then, *readex-dyn-detect* identifies the regions that are worth tuning, and determines the tuning potential of the application. We quantify the application dynamism w.r.t. two metrics: compute intensity and execution time. A configuration file contains the significant regions, the tuning parameters, the energy measurement plugin, and the tuning objective, and is used by PTF to perform DTA. We presented the *intraphase* and *interphase* tuning plugins that exploit the intra-phase and inter-phase dynamism respectively. The *intraphase* tuning plugin exploits the application layer of the HPC stack by tuning the ATPs, such as different decomposition algorithms, preconditioners or blocking factors. It then determines a single static-best configuration for all the phases, and individual rts-best configurations.

To exploit the inter-phase dynamism, we presented a set of metrics to characterize the phase behaviour. These metrics, known as *features* are PAPI hardware performance counters that are used to monitor events at the CPU level, and provide detailed insights on the application execution. The *interphase* tuning plugin uses these features to group phases with similar characteristics using DBSCAN to first group phases that are close to each other, and result in dense clusters. Then, the noise points from DBSCAN are analyzed by the spectral clustering algorithm to determine further associations or similarities using eigengaps to perform a graph cut. We emphasize that we set a hard limit on the minimum number of points in a cluster, and hence discard any clusters containing fewer points than the threshold.

While the *interphase* tuning plugin can potentially select the best configurations for the clusters at an early stage, we argue for the need for an additional tuning step to improve the confidence in the tuning result. Hence, we perform search space optimization using a targeted search of the tuning parameters by selecting a configuration to evaluate using a probabilistic random search strategy based on a Gaussian distribution. This is premised on the idea that certain configurations are better suited to reduce the normalized energy consumption of a cluster of phases, and thus, are more attractive for evaluation. The random strategy picks a configuration from a discrete probability distribution that is generated from the summation of individual Gaussian bell curves of the *attractor* and *repeller* configurations. The Gaussians are functions of the normalized energy consumption, and the distance between a configuration and an attractor or a repeller.

The best configurations for the phases and the rts's determined by the *intraphase* and the *interphase* tuning plugins are encapsulated in the tuning model. The tuning model guides the second stage of our approach, called Runtime Application Tuning (RAT). During RAT, the READEX Runtime Library (RRL) reads the tuning model, and dynamically switches the system configuration for the phase and the rts's. For inter-phase tuning, the tuning model additionally contains the ranges of the cluster features and the phases belonging to each cluster. The application is first linked with the runtime cluster prediction library, which uses three different cluster predictors based on a second-order Markov chain, one-bit and two-bit dynamic branch prediction schemes to identify the phase behavior at runtime by predicting the cluster id of an unseen phase.

We highlight that our approach can be used with minimal user involvement. The pre-analysis steps automatically filter overly fine-granular regions, and identify the dynamism. The tuning strategy results in a tuning model with automatic compiler instrumentation. We manually instrument different significant regions only to reduce the instrumentation overhead, a general problem for all instrumentation-based tools. Furthermore, our approach executes the application, in the worst case, two times during DTA. Moreover, these runs can be limited to a representative set of progress loop iterations instead of the entire application run.

We compared the theoretical energy savings computed during DTA with runtime savings while taking into account the switching overhead. We focused on three dynamic complex real-world applications, 128.GAP-geofem, sam(oa)^2 and INDEED, and one proxy benchmark, miniMD from the Mantevo project instead of benchmarks that usually call the same regions and exhibit the same behavior in each iteration. Maximum overall improvements of 18.61% for the job energy, 18.6% for CPU energy, and nearly 20% for performance were observed for sam(oa)^2 for multi-node runs, with the two-bit cluster predictor performing the best, followed by the second-order Markov chain predictor. The least savings were observed for the single node run of miniMD, with an improvement of less than 1% for the CPU energy for all three predictors. This is because a new cluster is formed every tenth phase due to the reneighbouring of the atoms. Since the ranges of the cluster features for the new cluster are not known to the predictors, they always predict the wrong cluster for an unseen phase. Moreover, the worst performance degradation was seen for the multi-node run of miniMD.

We can conclude that although dynamic tuning results in the improvement of energy-efficiency, in most cases, it degrades the performance. This can be attributed to two root causes: the instrumentation overhead from Score-P, and the switching overhead from dynamically switching the CPU and uncore frequencies by the RRL. We trade-off execution time for energy improvement, which typically results in higher execution times. On the other hand, our methodology reduces overheads by pre-computing best solutions at design-time and simply switching between the configurations at runtime. This reduces runtime overhead as expensive runtime search techniques are not applied. Moreover, our methodology is capable of tuning dynamic HPC applications, and shows the potential to scale to future Exascale systems.

8.1 Future Work

This thesis presented one of the several ways of optimizing the energy-efficiency of dynamic HPC applications by identifying the characteristic behaviour of different phases, and clustering them based on their behaviour. It also described a targeted or selective tuning step that increases the probability of randomly selecting potentially good configurations to evaluate. While these methods already provide valuable insights and result in energy savings, there are a number of promising opportunities for improvements. In this section, we outline four different research directions for future improvements.

Tuning multiple objectives using a Pareto set: Our work focused on tuning the energy consumption at the cost of increased execution time. This preference for one objective over others has traditionally been formulated as a single objective minimization problem in autotuning frameworks. However, the optimization of one objective may cause adverse consequences on the others. As we observed for most of our test applications, the improvement in the energy-efficiency typically degrades the performance or time-to-solution. On the other hand, increasing the speed of the processor to shorten the time-to-solution may lead to an increase in the chip temperatures, thereby increasing the risk of chip failures [137]. Thus, the goal of multi-objective optimization to find a set of solutions that improves two or more objective functions, and makes an acceptable trade-off between these objective functions instead of a single best solution [138].

A decision vector x_1 is said to be Pareto dominant over another vector x_2 if $x_1 \leq x_2$, meaning that x_1 is better or equal in all objectives than x_2 [139]. Since we do not want one solution dominating the others, the autotuning problem can be formulated as a multi-objective optimization, which results in a set of non-dominated solutions called a Pareto-optimal set, in which each solution is a trade-off among different conflicting objectives. The set of objective values corresponding to the Pareto set is referred to as the Pareto-optimal front.

Existing evolutionary and stochastic algorithms, as well as simplex-based methods such as the Nelder-Mead algorithm are popular multi-objective optimization methods. The Nelder-Mead algorithm is a downhill simplex method that approximates the local gradient of an underlying objective, and defines simplex transformations to help it move towards a local minimum value.

Specifying input identifiers for inter-phase tuning: Currently, the *interphase* tuning plugin tunes the application for a specific application input. However, we observe that the cluster analysis is highly dependent on the application input, such as the grid refinement level, the dimensions and properties of a sheet metal, including the material type. In Figure 1.1, INDEED was executed using the input set for a steel sheet of thickness 0.75 mm. A change in the material type completely changes the behaviour, and generates a different set of cluster-best and rts-specific best configurations. For example, Figure 8.1 illustrates the trend in the execution time across the phases of INDEED when executed using the input set for an aluminum alloy sheet of thickness 1.03 mm.

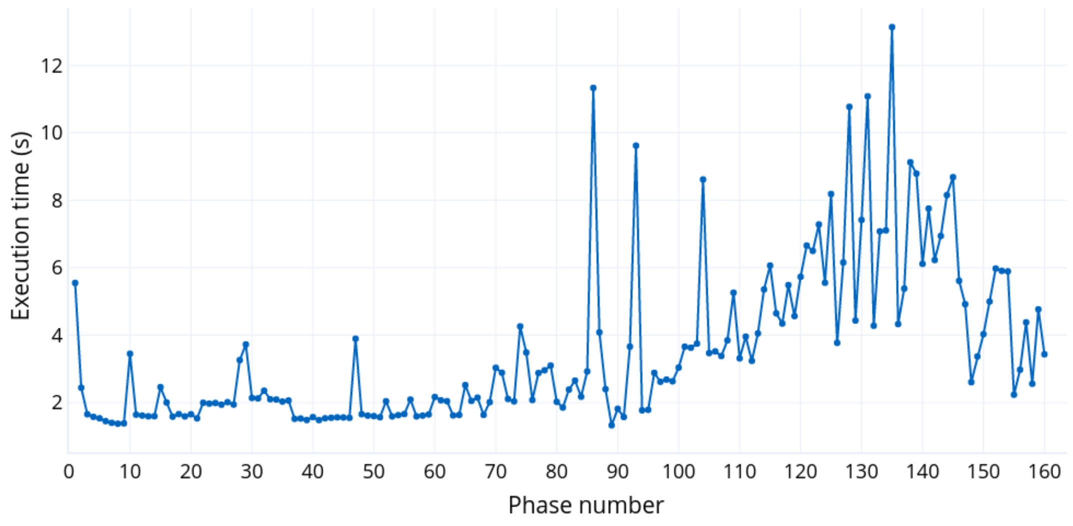


Figure 8.1: Variation of the execution time across the phases of INDEED when executed using an input set for an aluminum alloy sheet.

The above example indicates that the tuning results must be identified by specifying domain knowledge for the input identifiers, similar to the *intraphase* tuning plugin. The tuning model generation will then produce individual tuning models for each input, and merge the tuning models to generate the final application tuning model containing the cluster information for different input identifiers.

Runtime cluster prediction using neural networks:

The prediction of the behaviour of an unseen phase at runtime is a challenging task, since the requirements for runtime prediction are fixed. First, the predictor must be computationally cheap. Second, the predictions must be performed online at the beginning of a phase. Finally, predictions must be performed using the knowledge obtained from a small number of data points. In our work, identifying the behaviour of an unseen phase at runtime was performed by three simple cluster predictors.

The use of neural networks could potentially make the predictions more accurate. Among the neural networks are Recurrent Neural Networks (RNN) whose predictions are influenced by the historical knowledge. The aim of an RNN is to make use of sequential, non-linear outcomes to predict the present by using the memory about the information that has been computed so far. The memory of an RNN is considered as a hidden state, and the current outcome is calculated using the previous hidden state and the current input. An RNN repeatedly applies a transformation function to a series of inputs, and produces a series of output vectors as probabilities for each output. Thus, the previously computed outputs that are far away from the current time step do not contribute to the current prediction [140].

The advantage of RNNs is that they have a high prediction accuracy and efficiency in learning new patterns in the data. However, they work well only for larger input sizes. Thus, a possible solution would be to make the second-order Markov chain as the primary cluster predictor that runs in the foreground, and train an RNN in the background once a minimum number of phases have finished executing. After training the model, the output vector can be uploaded for the current run, and used to perform the prediction.

Comparing tuning results on newer Intel processors:

In our evaluation, we focused on optimizing the energy-efficiency for the Intel Haswell architecture. A future direction could be a comparison of the energy savings on newer Intel processors to leverage the new architectural implementations. First, from the five available C-states on Haswell processors, only four remain, i.e., C0, C1, and C6 on Skylake-SP processors. Second, the support for 512-bit wide vector operations were introduced in Skylake-SP, besides Xeon Phi [141].

Additionally, starting from the Broadwell architecture, the operating system can assign the control of the P-states to the processor and its internal power control unit to make autonomous decisions to switch the core frequency and uncore frequency based on internally collected execution statistics using Hardware Power Management (HWPM). HWPM is a new power saving technique that can be used to configure two operating modes: the native mode, based on enhanced Intel SpeedStep technology, and assigns the control of the P-states to the operating system, or HWPM, in which the power management is taken over by the processor [142]. While the Broadwell processors act autonomously, Skylake processors use a collaborative interface-based switching with the OS via interrupts [141], where the OS can define minimal, efficient and maximal frequencies for an execution.

Bibliography

[1] Parham Haririan. DVFS and Its Architectural Simulation Models for Improving Energy Efficiency of Complex Embedded Systems in Early Design Phase. *Computers*, 9, 2020. DOI: 10.3390/computers9010002.

[2] Christian Bischof, Dieter an Mey, and Christian Iwainsky. Brainware for Green HPC. *Computer Science — Research and Development*, 27:227–233, 2012. DOI: 10.1007/s00450-011-0198-5.

[3] Indeed: Highly Accurate Finite Element Simulation for Sheet Metal Forming. `http://gns-mbh.com/en/produkte/indeed/`.

[4] Yury Oleynik, Michael Gerndt, Joseph Schuchart, Per Gunnar Kjeldsberg, and Wolfgang E. Nagel. Run-Time Exploitation of Application Dynamism for Energy-Efficient Exascale Computing (READEX). In *18th International Conference on Computational Science and Engineering (CSE)*, pages 347–350. IEEE, Oct 2015. DOI: 10.1109/CSE.2015.55.

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

[6] Mohammed Sourouri, Espen Birger Raknes, Nico Reissmann, Johannes Langguth, Daniel Hackenberg, Robert Schöne, and Per Gunnar Kjeldsberg. Towards Fine-Grained Dynamic Tuning of HPC Applications on Modern Multi-Core Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17. Association for Computing Machinery, 2017. DOI: 10.1145/3126908.3126945.

[7] Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K. Hollingsworth, Boyana Norris, and Richard Vuduc. Autotuning in High-Performance Computing Applications. *Proceedings of the IEEE*, 106:2068–2083, Nov 2018. DOI: 10.1109/JPROC.2018.2841200.

[8] Michael Gerndt, Eduardo César, and Siegfried Benkner, editors. *Automatic Tuning of HPC Applications - The Periscope Tuning Framework*. Shaker Verlag, Aachen, 2015. ISBN: 978-3-8440-3517-9.

[9] Siegfried Benkner, Franz Franchetti, Hans Michael Gerndt, and Jeffrey K. Hollingsworth. Automatic Application Tuning for HPC Architectures (Dagstuhl Seminar 13401). *Dagstuhl Reports*, 3:214–244, 2014. DOI: 10.4230/DagRep.3.9.214.

[10] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer, 2012. DOI:10.1007/978-3-642-31476-6_7.

[11] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996.

[12] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On Spectral Clustering: Analysis and an Algorithm. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, pages 849–856, Cambridge, MA, USA, 2001. MIT Press.

[13] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, pages 157–173. Springer Berlin Heidelberg, 2010. DOI: 10.1007/978-3-642-11261-4_11.

[14] Michael Gerndt, Madhura Kumaraswamy, Nico Reissmann, Robert Schöne, Uldis Locans, and Per Gunnar Kjeldsberg. Final Computation of Configurations. *Deliverable 2.3 of the READEX project funded under the European Union's Horizon 2020 Programme EC GA No: 671657*, 2018. `https://www.readex.eu/wp-content/uploads/2018/05/D2_3.pdf`.

[15] Per Gunnar Kjeldsberg, Robert Schöne, Andreas Gocht, Umbreen Sabir Mian, and Nico Reissmann. Final mechanisms for run-time detection, switching, and calibration. *Deliverable 3.2 of the READEX project funded under the European Union's Horizon 2020 Programme EC GA No: 671657*, 2018. `https://www.readex.eu/wp-content/uploads/2018/05/D3_2.pdf`.

[16] Andreas Gocht, Robert Schöne, and Mario Bielert. Q-Learning Inspired Self-Tuning for Energy Efficiency in HPC. *arXiv preprint arXiv:1906.10970*, 2019. `http://arxiv.org/abs/1906.10970`.

[17] Jesus Carretero, Salvatore Distefano, Dana Petcu, Daniel Pop, Thomas Rauber, Gudula Runger, and David Singh. Energy-Efficient Algorithms for Ultrascale Systems. *Supercomput. Front. Innov.: Int. J.*, 2:77–104, April 2015. DOI: 10.14529/jsfi150205.

[18] Pavlos Petoumenos, Lev Mukhanov, Zheng Wang, Hugh Leather, and Dimitrios S. Nikolopoulos. Power Capping: What Works, What Does Not. In *Proceedings of the 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, ICPADS '15, page 525–534. IEEE Computer Society, 2015. DOI: 10.1109/ICPADS.2015.72.

[19] Pawel Czarnul, Jerzy Proficz, and Adam Krzywaniak. Energy-Aware High-Performance Computing: Survey of State-of-the-Art Tools, Techniques, and Environments. *Scientific Programming*, 2019, 2019. DOI: 10.1155/2019/8348791.

[20] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, USA, 4th edition, 2010.

[21] Luis Maeda-Nunez. *System-level power management using online machine learning for prediction and adaptation*. PhD thesis, University of Southampton, July 2016. `https://eprints.soton.ac.uk/399995/`.

[22] acpicpu"(4)" - NetBSD Manual Pages. `https://netbsd.gw.com/cgi-bin/man-cgi?acpicpu+4.i386+NetBSD-7.0.1`.

[23] Fabio Ferrero. Analysis and dynamic optimization of energy consumption on HPC applications based on real-time metrics, October 2017. `http://webthesis.biblio.polito.it/6423/`.

[24] 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 '16, page 69–76. IEEE Press, 2016. DOI: 10.1109/e2sc.2016.015.

[25] Hao Shen, Ying Tan, Jun Lu, Qing Wu, and Qinru Qiu. Achieving Autonomous Power Management Using Reinforcement Learning. *ACM Trans. Des. Autom. Electron. Syst.*, 18, 2013. DOI: 10.1145/2442087.2442095.

[26] Kapil Dev, Indrani Paul, and Wei Huang. A Framework for Evaluating Promising Power Efficiency Techniques in Future GPUs for HPC. In *Proceedings of the 24th High Performance Computing Symposium*, HPC '16. Society for Computer Simulation International, 2016. DOI: 10.22360/SpringSim.2016.HPC.003.

[27] Kapil Dev, Sherief Reda, Indrani Paul, Wei Huang, and Wayne P. Burleson. Workload-Aware Power Gating Design and Run-Time Management for Massively Parallel GPGPUs. *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 242–247, 2016. DOI: 10.1109/ISVLSI.2016.60.

[28] Sridutt Bhalachandra, Allan Porterfield, and Jan F. Prins. Using Dynamic Duty Cycle Modulation to Improve Energy Efficiency in High Performance Computing. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, IPDPSW '15, page 911–918. IEEE Computer Society, 2015. DOI: 10.1109/IPDPSW.2015.144.

[29] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B: System Programming Guide. `https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html`.

[30] Corey Gough, Ian Steiner, and Winston Saunders. *CPU Power Management*, pages 21–70. Apress, 2015. DOI: 10.1007/978-1-4302-6638-9_2.

[31] Chao Jin, Bronis R de Supinski, David Abramson, Heidi Poxon, Luiz DeRose, Minh Ngoc Dinh, Mark Endrei, and Elizabeth R. Jessup. A survey on software methods to improve the energy efficiency of parallel computing. *The International Journal of High Performance Computing Applications*, 31:517–549, 2017. DOI: 10.1177/1094342016665471.

[32] Vishal Gupta, Paul Brett, David Koufaty, Dheeraj Reddy, Scott Hahn, Karsten Schwan, and Ganapati Srinivasa. The Forgotten "Uncore": On the Energy-Efficiency of Heterogeneous Cores. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 34. USENIX Association, 2012.

[33] Vaibhav Sundriyal, Masha Sosonkina, Bryce M. Westheimer, and Mark Gordon. Comparisons of Core and Uncore Frequency Scaling Modes in Quantum Chemistry Application GAMESS. In *Proceedings of the High Performance Computing Symposium*, HPC '18. Society for Computer Simulation International, 2018.

[34] Nicholas P. Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua Fryman, Ivan Ganev, Roger A. Golliver, Rob Knauerhase, Richard Lethin, Benoit Meister, Asit K. Mishra, Wilfred R. Pinfold, Justin Teller, Josep Torrellas, Nicolas Vasilache, Ganesh Venkatesh, and Jianping Xu. Runnemede: An Architecture for Ubiquitous High-Performance Computing. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, page 198–209. IEEE Computer Society, 2013. DOI: 10.1109/HPCA.2013.6522319.

[35] Ananta Tiwari, Michael A. Laurenzano, Laura Carrington, and Allan Snavely. Auto-tuning for Energy Usage in Scientific Applications. In *Euro-Par 2011: Parallel Processing Workshops*, pages 178–187. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-29740-3_21.

[36] Cristian Tapus, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active Harmony: Towards Automated Performance Tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, page 1–11. IEEE Computer Society Press, 2002. DOI: 10.1109/SC.2002.10062.

[37] Josep Berral, Iñigo Goiri, Ramon Nou, Ferran Julià, J. Oriol Fitó, Jordi Guitart, Ricard Gavaldà, and Jordi Torres. *Toward Energy-Aware Scheduling Using Machine Learning*, pages 215–244. 07 2012. DOI: 10.1002/9781118342015.ch8.

[38] Li Yu, Zhou Zhou, Sean Wallace, Michael E Papka, and Zhiling Lan. Quantitative modeling of power performance tradeoffs on extreme scale systems. *Journal of Parallel and Distributed Computing*, 84:1–14, 2015. DOI: 10.1016/j.jpdc.2015.06.006.

[39] Yiannis Georgiou, Thomas Cadeau, David Glesser, Danny Auble, Morris Jette, and Matthieu Hautreux. Energy Accounting and Control with SLURM Resource and Job Management System. In *Distributed Computing and Networking*, pages 96–118. Springer Berlin Heidelberg, 2014. DOI: 10.1007/978-3-642-45249-9_7.

[40] Dineshkumar Rajagopal, Daniele Tafani, Yiannis Georgiou, David Glesser, and Michael Ott. A Novel Approach for Job Scheduling Optimizations Under Power Cap for ARM and Intel HPC Systems. *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 142–151, 2017. DOI: 10.1109/HiPC.2017.00025.

[41] Zhou Zhou, Zhiling Lan, Wei Tang, and Narayan Desai. Reducing Energy Costs for IBM Blue Gene/P via Power-Aware Job Scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 96–115. Springer Berlin Heidelberg, 2014. DOI: 10.1007/978-3-662-43779-7_6.

[42] Axel Auweter, Arndt Bode, Matthias Brehm, Luigi Brochard, Nicolay Hammer, Herbert Huber, Raj Panda, Francois Thomas, and Torsten Wilde. A Case Study of Energy Aware Scheduling on SuperMUC. In *Supercomputing*, pages 394–409. Springer International Publishing, 2014. DOI: 10.1007/978-3-319-07518-1_25.

[43] Aniruddha Marathe, Peter E. Bailey, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. A Run-Time System for Power-Constrained HPC Applications. In *High Performance Computing*, pages 394–408, 2015. DOI: 10.1007/978-3-319-20119-1_28.

[44] 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 '13, page 173–182. Association for Computing Machinery, 2013. DOI: 10.1145/2464996.2465009.

[45] Osman Sarood, Akhil Langer, Laxmikant Kale, Barry Rountree, and Bronis Supinski. Optimizing power allocation to CPU and memory subsystems in overprovisioned HPC systems. pages 1–8, 2013. DOI: 10.1109/CLUSTER.2013.6702684.

[46] Neha Gholkar, Frank Mueller, and Barry Rountree. Power Tuning HPC Jobs on Power-Constrained Systems. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, page 179–191. Association for Computing Machinery, 2016. DOI: 10.1145/2967938.2967961.

[47] Connor Imes, Huazhe Zhang, Kevin Zhao, and Henry Hoffmann. Handing dvfs to hardware: Using power capping to control software performance. *Technical Report TR-2018-03*, 2018. `https://newtraell.cs.uchicago.edu/files/tr_authentic/TR-2018-03.pdf`.

[48] Kazuki Tsuzuku and Toshio Endo. Power Capping of CPU-GPU Heterogeneous Systems using Power and Performance Models. *SMARTGREENS 2015 - 4th International Conference on Smart Cities and Green ICT Systems, Proceedings*, pages 226–233, 2015. DOI: 10.5220/0005445102260233.

[49] Cristina Silvano, Giovanni Agosta, Stefano Cherubin, Davide Gadioli, Gianluca Palermo, Andrea Bartolini, Luca Benini, Jan Martinovič, Martin Palkovič, Kateřina Slaninová, et al. The ANTAREX approach to autotuning and adaptivity for energy efficient HPC systems. In *Proceedings of the ACM International Conference on Computing Frontiers*, Proceedings of the ACM International Conference on Computing Frontiers, pages 288–293. ACM, 2016. DOI:10.1145/2903150.2903470.

[50] Daniele Cesarini, Andrea Bartolini, Piero Bonfà, Carlo Cavazzoni, and Luca Benini. COUNTDOWN: a run-time library for application-agnostic energy saving in MPI communication primitives. In *Proceedings of the 2nd Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems*, pages 1–6. Association for Computing Machinery, 2018. DOI: 10.1145/3295816.3295818.

[51] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase Tracking and Prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, page 336–349. Association for Computing Machinery, 2003. DOI: 10.1145/859618.859657.

[52] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. Trace Based Phase Prediction for Tightly-Coupled Heterogeneous Cores. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 445–456. Association for Computing Machinery, 2013. DOI: 10.1145/2540708.2540746.

[53] 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*, 2017. DOI: 10.1007/s00607-016-0532-7.

[54] Bilge Acun, Kavitha Chandrasekar, and Laxmikant V. Kalé. Fine-Grained Energy Efficiency Using Per-Core DVFS with an Adaptive Runtime System. *2019 Tenth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, 2019. DOI: 10.1109/IGSC48788.2019.8957174.

[55] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, page 359–370. IEEE Computer Society, 2006. DOI: 10.1109/MICRO.2006.30.

[56] Andreas Sembrant, David Eklov, and Erik Hagersten. Efficient Software-Based Online Phase Classification. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, page 104–115. IEEE Computer Society, 2011. DOI: 10.1109/IISWC.2011.6114207.

[57] Priya Nagpurkar, Chandra Krintz, Michael Hind, Peter F. Sweeney, and V. T. Rajan. Online Phase Detection Algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, page 111–123. IEEE Computer Society, 2006. DOI: 10.1109/CGO.2006.26.

[58] Jinpyo Kim, Sreekumar V. Kodakara, Wei-Chung Hsu, David J. Lilja, and Pen-Chung Yew. Dynamic Code Region (DCR) Based Program Phase Tracking and Prediction for Dynamic Optimizations. In *High Performance Embedded Architectures and Compilers*, pages 203–217. Springer Berlin Heidelberg, 2005. DOI: 10.1007/11587514_14.

[59] Juan Gonzalez, Judit Gimenez, and Jesus Labarta. Automatic Detection of Parallel Applications Computation Phases. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, IPDPS '09, page 1–11. IEEE Computer Society, 2009. DOI: 10.1109/IPDPS.2009.5161027.

[60] Xia Zhang, Xusheng Xiao, Liang He, Yun Yong Ma, Yangyang Huang, Xuanzhe Liu, Wenyao Xu, and Cong Liu. PIFA: An Intelligent Phase Identification and Frequency Adjustment Framework for Time-Sensitive Mobile Computing. *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 54–64, 2019. DOI: 10.1109/RTAS.2019.00013.

[61] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 45–57. Association for Computing Machinery, 2002. DOI: 10.1145/605397.605403.

[62] Kelly Livingston, Nicolas Triquenaux, Thibault Fighiera, Jean Christophe Beyler, and William Jalby. Computer using too much power? Give it a REST (Runtime Energy Saving Technology). *Computer Science-Research and Development*, 29:123–130, 2014. DOI: 10.1007/s00450-012-0226-0.

[63] Joshua Dennis Booth, Jagadish Kotra, Hui Zhao, Mahmut T. Kandemir, and Padma Raghavan. Phase Detection with Hidden Markov Models for DVFS on Many-Core Processors. In *35th IEEE International Conference on Distributed Computing Systems, ICDCS*, pages 185–195. IEEE Computer Society, 2015. DOI: 10.1109/ICDCS.2015.27.

[64] Jonathan Eastep, Steve Sylvester, Christopher Cantalupo, Brad Geltz, Federico Ardanaz, Asma Al-Rawi, Kelly Livingston, Fuat Keceli, Matthias Maiterth, and Siddhartha Jana. Global Extensible Open Power Manager: A Vehicle for HPC Community Collaboration on Co-Designed Energy Management Solutions. In *International Supercomputing Conference*, pages 394–412, 2017. DOI: 10.1007/978-3-319-58667-0_21.

[65] Robert Schöne and Daniel Molka. Integrating Performance Analysis and Energy Efficiency Optimizations in a Unified Environment. *Comput. Sci.*, 29:231–239, 2014. DOI: 10.1007/s00450-013-0243-7.

[66] Yoshihiko Hotta, Mitsuhisa Sato, Hideaki Kimura, Satoshi Matsuoka, Taisuke Boku, and Daisuke Takahashi. Profile-Based Optimization of Power Performance by Using Dynamic Voltage Scaling on a PC Cluster. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, page 298. IEEE Computer Society, 2006. DOI: 10.1109/IPDPS.2006.1639597.

[67] Karunakar Reddy Basireddy. *Runtime energy management of concurrent applications for multi-core platforms*. PhD thesis, University of Southampton, April 2019. `https://eprints.soton.ac.uk/433546/`.

[68] Ghislain Landry Tsafack Chetsa, Laurent Lefevrem, and Patricia Stolf. A three step blind approach for improving high performance computing systems' energy performance. *Concurrency and Computation: Practice and Experience*, 26:2612–2629, 2014. DOI: 10.1002/cpe.3312.

[69] Ghislain Landry Tsafack, Laurent Lefevre, Jean-Marc Pierson, Patricia Stolf, and Georges Da Costa. Exploiting Performance Counters to Predict and Improve Energy Performance of HPC Systems. *Future Generation Computer Systems*, 36:287–298, 2014. DOI: 10.1016/j.future.2013.07.010.

[70] Michael Gerndt and Michael Ott. Automatic performance analysis with periscope. *Concurrency and Computation: Practice and Experience*, 22:736–748, 2010. DOI: 10.1002/cpe.1551.

[71] Carla Guillen, Carmen Navarrete, David Brayford, Wolfram Hesse, and Matthias Brehm. DVFS automatic tuning plugin for energy related tuning objectives. In *2nd International Conference on Green High Performance Computing (ICGHPC)*, pages 1–8, 2016. DOI: 10.1109/ICGHPC.2016.7508061.

[72] Carmen B. Navarrete, Carla Guillén, Wolfram Hesse, and Matthias Brehm. Autotuning the energy consumption. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE), Proceedings of the International Conference on Parallel Computing, ParCo 2013*, pages 668–677, 2013. DOI:10.3233/978-1-61499-381-0-668.

[73] Robert Springer, David K. Lowenthal, Barry Rountree, and Vincent W. Freeh. Minimizing execution time in MPI programs on an energy-constrained, power-scalable cluster. Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP, pages 230–238, 2006. DOI: 10.1145/1122971.1123006.

[74] Joshua Peraza, Ananta Tiwari, Michael Laurenzano, Laura Carrington, and Allan Snavely. PMaC's Green Queue: A Framework for Selecting Energy Optimal DVFS Configurations in Large Scale MPI Applications. *Concurrency and Computation: Practice and Experience*, 28:211–231, 2016. DOI: 10.1002/cpe.3184.

[75] Karthik Elangovan, Ivan Rodero, Manish Parashar, Francesc Guim, and Isaac Hernandez. Adaptive memory power management techniques for HPC workloads. pages 1–11, 2011. DOI: 10.1109/HiPC.2011.6152740.

[76] Chung-hsing Hsu and Wu-chun Feng. A Power-Aware Run-Time System for High-Performance Computing. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05. IEEE Computer Society, 2005. DOI: 10.1109/SC.2005.3.

[77] Thomas Rauber, Gudula Rünger, and Matthias Stachowski. Model-based optimization of the energy efficiency of multi-threaded applications. In *Eighth International Green and Sustainable Computing Conference (IGSC)*, pages 1–6, 2017. DOI: 10.1109/IGCC.2017.8323578.

[78] Bo Su, Junli Gu, Li Shen, Wei Huang, Joseph L. Greathouse, and Zhiying Wang. PPEP: Online Performance, Power, and Energy Prediction Framework and DVFS Space Exploration. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, page 445–457. IEEE Computer Society, 2014. DOI: 10.1109/MICRO.2014.17.

[79] Ruben Vazquez, Ann Gordon-Ross, and Greg Stitt. Machine Learning-based Prediction for Dynamic, Runtime Architectural Optimizations of Embedded Systems. In *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pages 1–7, 2019. DOI: 10.1109/NORCHIP.2019.8906901.

[80] Ryan Cochran, Can Hankendi, Ayse Coskun, and Sherief Reda. Identifying the Optimal Energy-Efficient Operating Points of Parallel Workloads. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '11, page 608–615. IEEE Press, 2011. DOI: 10.1109/ICCAD.2011.6105393.

[81] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. Pack & Cap: Adaptive DVFS and Thread Packing under Power Caps. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 175–185. Association for Computing Machinery, 2011. DOI: 10.1145/2155620.2155641.

[82] Gence Ozer, Sarthak Garg, Neda Davoudi, Gabrielle Poerwawinata, Matthias Maiterth, Alessio Netti, and Daniele Tafani. Towards a Predictive Energy Model for HPC Runtime Systems Using Supervised Learning. In *Proc. of PMACS Workshop*, 2019. DOI: 10.1007/978-3-030-48340-1_48.

[83] Shervin Hajiamini and Behrooz Shirazi. Evaluation of a Practical Markov model-based Methodology for Energy Efficiency in Multicore Systems. *Tenth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, 2019. DOI: 10.1109/IGSC48788.2019.8957190.

[84] Shajulin Benedict, R.S. Rejitha, Philipp Gschwandtner, Radu Prodan, and Thomas Fahringer. Energy Prediction of OpenMP Applications Using Random Forest Modeling Approach. *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 1251–1260, 2015. DOI: 10.1109/IPDPSW.2015.12.

[85] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996. DOI: 10.1613/jair.301.

[86] Barry Rountree, David K. Lowenthal, 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 '09, page 460–469. Association for Computing Machinery, 2009. DOI: 10.1145/1542275.1542340.

[87] Min Yeol Lim, Vincent W. Freeh, and David K. Lowenthal. Adaptive, Transparent Frequency and Voltage Scaling of Communication Phases in MPI Programs. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, page 107. Association for Computing Machinery, 2006. DOI: 10.1145/1188455.1188567.

[88] Vincent W. Freeh and David K. Lowenthal. Using Multiple Energy Gears in MPI Programs on a Power-Scalable Cluster. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, page 164–173. Association for Computing Machinery, 2005. DOI: 10.1145/1065944.1065967.

[89] Etienne André, Remi Dulong, Amina Guermouche, and François Trahay. DUF : Dynamic Uncore Frequency scaling to reduce power consumption. working paper or preprint. `https://hal.archives-ouvertes.fr/hal-02401796/file/report.pdf`.

[90] Neha Gholkar, Frank Mueller, and Barry Rountree. Uncore Power Scavenger: A Runtime for Uncore Power Conservation on HPC Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19. Association for Computing Machinery, 2019. DOI: 10.1145/3295500.3356150.

[91] Pietro Cicotti, Ananta Tiwari, and Laura Carrington. Efficient speed (ES): Adaptive DVFS and clock modulation for energy efficiency. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 158–166. IEEE, 2014. DOI: 10.1109/CLUSTER.2014.6968750.

[92] Reiji Suda. A Bayesian method of online automatic tuning. In *Software Automatic Tuning*, pages 275–293. Springer, 2011. DOI: 10.1007/978-1-4419-6935-4_16.

[93] Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. Optimizing I/O Performance of HPC Applications with Autotuning. *ACM Trans. Parallel Comput.*, 5, 2019. DOI: 10.110.1145/3309205.

[94] Madhura Kumaraswamy, Anamika Chowdhury, and Michael Gerndt. Design-Time Analysis for the READEX Tool Suite. In *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing, ParCo 2017*, pages 307–316, 2017. DOI: 10.3233/978-1-61499-843-3-307.

[95] PAPI: Performance Application Programming Interface. `http://icl.cs.utk.edu/papi/`.

[96] Heike Jagode, Anthony Danalis, and Jack Dongarra. What it Takes to keep PAPI Instrumental for the HPC Community. In *1st Workshop on Sustainable Scientific Software (CW3S19)*, 2019. `https://collegeville.github.io/CW3S19/WorkshopResources/WhitePapers/JagodeHeike_CW3S19_papi.pdf`.

[97] Madhura Kumaraswamy, Anamika Chowdhury, Michael Gerndt, Zakaria Bendifallah, Othman Bouizi, Uldis Locans, Lubomír Říha, Ondřej Vysocký, Martin Beseda, and Jan Zapletal. Domain knowledge specification for energy tuning. *Concurrency and Computation: Practice and Experience*, 31, 2019. DOI: 10.1002/cpe.4650.

[98] Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stelios Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, and Koen De Bosschere. System-Scenario-Based Design of Dynamic Embedded Systems. *ACM Trans. Des. Autom. Electron. Syst.*, 14, 2009. DOI: 10.1145/1455229.1455232.

[99] Evan Rosser, Wayne Kelly, Bill Pugh, Dave Wonnacott, Tatiana Shpeisman, and Vadim Maslov. The Omega Calculator. `https://github.com/davewathaverford/the-omega-project`.

[100] J. Chang, K.B. Nakshatrala, M.G. Knepley, and L. Johnsson. A performance spectrum for parallel computational frameworks that solve PDEs. *Concurrency and Computation: Practice and Experience*, 30. DOI: 10.1002/cpe.4401.

[101] Thomas Röhl, Jan Eitzinger, Georg Hager, and Gerhard Wellein. Validation of Hardware Events for Successful Performance Pattern Identification in High Performance Computing. In *Tools for High Performance Computing 2015*, pages 17–28. Springer International Publishing, 2016. DOI: 10.1007/978-3-319-39589-0_2.

[102] Bhavishya Goel. *Measurement, Modeling, and Characterization for Energy-efficient Computing*. PhD thesis, Chalmers University of Technology, 2016. `https://research.chalmers.se/publication/236780/file/236780_Fulltext.pdf`.

[103] Alexander Strehl and Joydeep Ghosh. *Relationship-Based Clustering and Cluster Ensembles for High-Dimensional Data Mining*. PhD thesis, 2002. `http://hdl.handle.net/2152/967`.

[104] Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications. *Data Min. Knowl. Discov.*, 2:169–194, June 1998. DOI:10.1023/A:1009745219419.

[105] Manisha Naik Gaonkar and Kedar Sawant. AutoEpsDBSCAN: DBSCAN with Eps automatic for large dataset. *International Journal on Advanced Computer Theory and Engineering*, 2:11–16, 2013.

[106] Madhura Kumaraswamy and Michael Gerndt. Leveraging Inter-Phase Application Dynamism for Energy-Efficiency Auto-tuning. In *PDPTA'18: The 24th International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 132–138, 2018. `https://csce.ucmss.com/cr/books/2018/LFS/CSREA2018/PDP3694.pdf`.

[107] Ulrike Luxburg. A Tutorial on Spectral Clustering. *Statistics and Computing*, 17:395–416, 2007. DOI: 110.1007/s11222-007-9033-z.

[108] S. Chen, Z. Li, X. Tang, and J. Liu. Noise Robust Spectral Clustering. In *2007 11th IEEE International Conference on Computer Vision*, pages 1–8. IEEE Computer Society, 2007. DOI: 10.1109/ICCV.2007.4409061.

[109] Yuan Wang, Xiaochun Wang, and Xia Li Wang. A Spectral Clustering Based Outlier Detection Technique. In *Machine Learning and Data Mining in Pattern Recognition*, pages 15–27. Springer International Publishing, 2016. DOI: 10.1007/978-3-319-41920-6_2.

[110] Lihi Zelnik-Manor and Pietro Perona. Self-Tuning Spectral Clustering. In *Proceedings of the 17th International Conference on Neural Information Processing Systems*, NIPS'04, page 1601–1608, Cambridge, MA, USA, 2004. MIT Press.

[111] Francis R. Bach and Michael I. Jordan. Learning Spectral Clustering, With Application To Speech Separation. *J. Mach. Learn. Res.*, 7:1963–2001, December 2006.

[112] Nurul Nnadiah Zakaria, Mahmod Othman, Rajalingam Sokkalingam, Hanita Daud, Lazim Abdullah, and Evizal Abdul Kadir. Markov Chain Model Development for Forecasting Air Pollution Index of Miri, Sarawak. *Sustainability*, 11:5190, 2019. DOI: 10.3390/su11195190.

[113] Bruce A. Craig and Peter P. Sendi. Estimation of the transition matrix of a discrete-time Markov chain. *Health Economics*, 11:33–42, 2002. DOI: 10.1002/hec.654.

[114] Charles Miller Grinstead and James Laurie Snell. *Introduction to probability*. American Mathematical Soc., 2012. `https://math.dartmouth.edu/~prob/prob/prob.pdf`.

[115] Giorgio Alfredo Spedicato. Discrete Time Markov Chains with R. *R Journal*, 9, 2017. DOI: 10.32614/RJ-2017-036.

[116] Dilek Eren Akyuz, Mehmetcik Bayazit, and Bihrat Onoz. Markov chain models for hydrological drought characteristics. *Journal of Hydrometeorology*, 13:298–309, 2012. DOI: 10.1175/JHM-D-11-019.1.

[117] Chuan Zhang, Ziyuan Shen, Wei Wei, Jing Zhao, Zaichen Zhang, and Xiaohu You. Molecular computing for Markov chains. *Natural Computing*, pages 1–16, 2019. DOI: 10.1007/s11047-019-09736-8.

[118] David H. Bailey. The NAS Parallel Benchmarks. In *Encyclopedia of Parallel Computing*, pages 1254–1259. Springer, 2011. DOI: 10.1007/978-0-387-09766-4.

[119] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52:65–76, 2009. DOI: 10.1145/1498765.1498785.

[120] Joseph Schuchart, Michael Werner, Andreas Gocht, and Sven Schiffner. Score-P x86 Energy plugin. `https://github.com/readex-eu/scorep_plugin_x86_energy`.

[121] Madhura Kumaraswamy and Michael Gerndt. Exploiting Dynamism in HPC Applications to Optimize Energy-Efficiency. In *49th International Conference on Parallel Processing - ICPP : Workshops (ICPP Workshops '20)*, ICPP Workshops '20, New York, NY, USA, 2020. Association for Computing Machinery. DOI: 10.1145/3409390.3409399.

[122] Madhura Kumaraswamy, Anamika Chowdhury, Andreas Gocht, Jan Zapletal, Kai Diethelm, Lubomir Riha, Marie-Christine Sawley, Michael Gerndt, Nico Reissmann, Ondrej Vysocky, Othman Bouizi, Per Gunnar Kjeldsberg, Ramon Carreras, Robert Schöne, Umbreen Sabir Mian, Venkatesh Kannan, and Wolfgang E. Nagel. Saving Energy Using the READEX Methodology. In *International Workshop on Parallel Tools for High Performance Computing*. Springer, 2018. Accepted for publication.

[123] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring Energy Consumption for Short Code Paths Using RAPL. *SIGMETRICS Perform. Eval. Rev.*, 40:13–17, 2012. DOI: 10.1145/2425248.2425252.

[124] Standard Performance Evaluation Corporation. 128.GAPgeofem SPEC MPI2007 Benchmark Description. `https://www.spec.org/auto/mpi2007/Docs/128.GAPgeofem.html`.

[125] Hiroshi Okuda, Kengo Nakajima, Mikio Iizuka, Li Chen, and Hisashi Nakamura. Parallel Finite Element Analysis Platform for the Earth Simulator: GeoFEM. volume 2659, pages 773–780, 01 2003. DOI: 10.1007/3-540-44863-2_75.

[126] Matthias S Müller, Matthijs Van Waveren, Ron Lieberman, Brian Whitney, Hideki Saito, Kalyan Kumaran, John Baron, William C Brantley, Chris Parrott, Tom Elken, et al. SPEC MPI2007—an application benchmark suite for parallel systems using MPI. *Concurrency and Computation: Practice and Experience*, 22:191–205, 2010. DOI: 10.1002/cpe.1535.

[127] Oliver Meister. Sam(oa)^2 - SFCs and Adaptive Meshes for Oceanic And Other Applications. `https://github.com/meistero/Samoa`.

[128] Ao Mo-Hellenbrand. *Resource-Aware and Elastic Parallel Software Development for Distributed-Memory HPC Systems*. Dissertation, Technische Universität München, München, 2019.

[129] Sebastian Rettenberger. ASAGI: a pArallel Server for Adaptive GeoInformation. `https://github.com/TUM-I5/ASAGI`.

[130] Alexander Breuer and Michael Bader. Teaching Parallel Programming Models on a Shallow-Water Code. *2012 11th International Symposium on Parallel and Distributed Computing*, pages 301–308, 2012. DOI: 10.1109/ISPDC.2012.48.

[131] Oliver Meister. *Sierpinski Curves for Parallel Adaptive Mesh Refinement in Finite Element and Finite Volume Methods*. Dissertation, Technische Universität München, München, 2016.

[132] British Oceanographic Data Centre. General Bathymetric Chart of the Oceans. `https://www.gebco.net/`.

[133] Kai Diethelm. Tools for assessing and optimizing the energy requirements of high performance scientific computing software. *PAMM*, 16:837–838, 05 2016. DOI: 10.1002/pamm.201610407.

[134] GitHub - Mantevo/miniMD: MiniMD Molecular Dynamics Mini-App. `https://github.com/Mantevo/miniMD`.

[135] Charm++: Mini-Apps. `http://charmplusplus.org/miniApps/#leanmd`.

[136] John Pennycook and Stephen A. Jarvis. Developing Performance-Portable Molecular Dynamics Kernels in OpenCL. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 386–395, 2012. DOI: 10.1109/SC.Companion.2012.58.

[137] Herbert Jordan, Peter Thoman, Juan J Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A multi-objective auto-tuning framework for parallel codes. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2012. DOI: 10.1109/SC.2012.7.

[138] Mohammad Azzeh, Ali Bou Nassif, Shadi Banitaan, and Fadi Almasalha. Pareto efficient multi-objective optimization for local tuning of analogy-based estimation. *Neural Computing and Applications*, 27:2241–2265, 2016. DOI: 10.1007/s00521-015-2004-y.

[139] Antoine S Dymond, Andries P Engelbrecht, Schalk Kok, and P Stephan Heyns. Tuning optimization algorithms under multiple objective function evaluation budgets. *IEEE Transactions on Evolutionary Computation*, 19:341–358, 2014. DOI: 110.1109/TEVC.2014.2322883.

[140] Gang Chen. A gentle tutorial of recurrent neural network with error backpropagation. *arXiv preprint arXiv:1610.02583*, 2016. `https://arxiv.org/abs/1610.02583`.

[141] Robert Schöne, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. Energy efficiency features of the Intel Skylake-SP processor and their impact on performance. *arXiv preprint arXiv:1905.12468*, 2019. `https://arxiv.org/abs/1905.12468`.

[142] Fujitsu. BIOS optimizations for Xeon Scalable processors based systems. July 2019. `https://sp.ts.fujitsu.com/dmsp/Publications/public/wp-skylake-RXTX-bios-settings-primergy-ww-en.pdf`.