200 PFlop/s. To reach the exascale level with this technology would already require 48 MW power. Therefore, energy reduction is a major goal for hardware, OS, runtime system, application and tool developers.

The European Horizon 2020 project READEX (Runtime Exploitation of Application Dynamism for Energy-efficient eXascale Computing)[8] funded from 2015-2018 developed the READEX tool suite[9] for dynamic energy-efficiency tuning of applications. It semi-automatically tunes hybrid HPC applications by splitting the tuning into a *Design-Time Analysis* (DTA) and a *Runtime Application Tuning* (RAT) phase.

READEX configures different tuning knobs (hardware, software and application parameters) based on dynamic application characteristics. Clock frequencies have a significant impact on performance and power consumption of a processor and therefore to the energy efficiency of the system. To leverage this potential, Intel Haswell processors allow to change core and the uncore frequencies. One basic idea is to lower the core frequency and increase the uncore frequency for memory bound regions since the processor is anyway waiting for data from memory. Another important tuning parameter is the number of parallel OpenMP threads in a node. If a routine is memory bound, it might be more energy-efficient to use only so many threads until the memory bandwidth is saturated [30].

Following the scenario-based tuning approach [9] from the embedded world, READEX creates an *Application Tuning Model* (ATM) at design-time, which specifies optimal configurations of the tuning parameters for individual program regions. This tuning model is then passed to RAT and is applied during production runs of the application by switching the tuning parameters to their optimal values when a tuned region is encountered. This dynamic tuning approach of READEX allows to exploit dynamic changes in the application characteristics for energy tuning while static approaches only optimize the settings for the entire program run.

READEX even goes beyond tuning individual regions. It can distinguish different instances of regions, e.g., resulting from different call sites in the code. These instances, called *runtime situations* (rts), can have different optimal configurations in the ATM.

READEX leverages well established tools, such as Score-P for instrumentation and monitoring of the application [17]. The *READEX Runtime Library* (RRL) implements RAT and is a new plugin for Score-P. The DTA is implemented via new tuning plugins for the Periscope Tuning Framework [24].

For reading energy measurements and for modifying core and uncore frequencies, several interfaces are supported, such as libMSRsafe [16] and LIKWID [32]. Power measurements are based on the RAPL [6] counters or on special hardware, such as HDEEM [13] on the Taurus machine in Dresden. If the target machine provides such special hardware, new measurement plugins for Score-P have to be implemented. Moreover, reading energy measurements as well as setting the frequencies can require root privileges and appropriate extensions to the ker-

[8] www.readex.eu

[9] We use the abbreviation READEX for the tool suite in the rest of the paper.

nel of the machine. The availability of these interfaces is currently the major limitation for applying the READEX tool suite.

This paper focuses on the features of READEX, the steps to be taken by the user in applying the tools, and motivates why certain aspects are more or less suited for different applications. The paper is neither a user's guide nor a technical documentation of the inner workings of the tools comprising the tool suite.

Section 2 presents related available tools for energy management of HPC systems. Section 3 introduces the major steps in using the READEX tool suite and presents Pathway [25], a tool for automating tool workflows for HPC systems. Section 4 presents DTA as the first step in tuning applications. The reasoning behind the generation of the ATM at the end of DTA is given in Section 5. Section 6 introduces a tool for the visual analysis of the generated ATM. RAT is presented in Section 7, and the visualization of dynamic switching based on the given ATM in Vampir is introduced in Section 8. Extensions to the READEX tool suite are presented in Section 9, and results from several applications on different machines are summarized in Section 10. Finally, the paper draws some conclusions.

2 State-of-the-Art

As energy-efficiency and consumption have now become one of the biggest challenges in HPC, research in this direction has gained momentum. Currently, there are many approaches that employ DVFS in HPC to tune different objectives. Rojek et al. [27] used DVFS to reduce the energy consumption for stencil computations whose execution time is predefined. The algorithm first collects the execution time and energy for a subset of the processor frequencies, and dynamically models the execution time and the average energy consumption as functions of the operational frequency. It then adjusts the frequency so that the predefined execution time is respected.

not defined here but on p. 8

Imes et al. [15] developed machine learning classifiers to tune socket resource allocation, HyperThreads and DVFS to minimize energy. The classifiers predict different system settings using the measurements obtained by polling performance counters during the application run. This approach cannot be used for dynamic applications because of the overhead from the classifier in predicting new settings due to rapid fluctuations in the performance counters. READEX, however, focuses on auto-tuning dynamic applications.

The ANTAREX project [31] specifies adaptivity strategies, parallelization and mapping of the application at runtime by using a Domain Specific Language (DSL) approach. During design-time, control code is added to the application to provide runtime monitoring strategies. At runtime, the framework performs auto-tuning by configuring software knobs for application regions using the runtime information of the application execution. This approach is specialized for ARM-based multi-cores and accelerators, while READEX targets all HPC systems.

Intel's open-source Global Extensible Open Power Manager (GEOPM) [7] runtime framework provides a plugin based energy management system for power-constrained systems. GEOPM supports both offline and online analysis by sampling performance counters, identifying the nodes on the critical path, estimating the power consumption and then adjusting the power budget among these nodes. GEOPM adjusts individual power caps for the nodes instead of a uniform power capping by allocating a larger portion of the job power budget to the critical path.

Conductor [23], a runtime system used at the Lawrence Livermore National Laboratory also performs adaptive power balancing for power capped systems. It first performs a parallel exploration of the configuration space by setting a different thread concurrency level and DVFS configuration on each MPI process, and statically selects the optimal configuration for each task. It then performs adaptive power balancing to distribute more power to the critical path.

The AutoTune project [12, 11] developed a DVFS tuning plugin to auto-tune energy consumption, total cost of ownership, energy delay product and power capping. The tuning is performed using a model that predicts the energy and power consumption as well as the execution time at different CPU frequencies. It uses the enopt library to vary the frequency for different application regions. While this is a static approach, READEX implements dynamic tuning for rts's.

READEX goes beyond previous works by tuning the uncore frequency. It also exploits the dynamism that exists between individual iterations of the main progress loop.

3 The READEX Methodology

The READEX methodology is split into two phases: design-time (during application development) and runtime (during production runs). READEX performs a sequence of steps to produce tuning advice for an application. The following sections describe the steps defined in the READEX methodology.

3.1 Application Instrumentation and Analysis Preparation

The first step in READEX is to instrument the HPC application by inserting probe-functions around different regions that are of interest to tuning. A region can be any arbitrary part of the code, for instance a function or a loop. READEX is based on instrumentation with Score-P and requires that the *phase region* be manually annotated. A phase region is a single-entry and exit region where most of the application progresses. Typically, a phase region may be the body of the main progress loop of the application. In addition to the phase region, READEX supports instrumentation of *user regions* inside the phase region. User regions may be instrumented manually using Score-P or automatically by the compiler.

The READEX methodology also allows exposing parameters that define the dynamic behaviour of the application. These parameters, referred to as additional identifiers, will enhance the distinction of different scenarios into runtime

situations during the application execution, potentially leading to better identification of optimal configurations for the different tuning parameters. An example of additional identifiers is different input data sets. Exposing this information allows READEX to detect different compute characteristics of the application caused by different inputs.

3.2 Application Pre-analysis

After instrumenting an application and preparing it for analysis, the second step in the READEX approach is to perform a pre-analysis. The objective of this step is to automatically identify and characterize dynamism in the application behaviour. This is critical because the READEX approach is based on tuning hardware, system and application parameters, depending on the dynamism exhibited by the different regions in the application. READEX is capable of identifying and characterizing two types of application dynamism:

- *Inter-phase dynamism*: This occurs when each phase (execution instance) of a phase region in the application exhibits different characteristics. This results in different values for the measured objective values (execution time) and thus may require different configurations to be applied for the tuning parameters.
- *Intra-phase dynamism*: This occurs when each runtime situation (execution instance) of the significant regions in a phase region exhibits different characteristics and results in different values for the measured objective values (execution time, compute intensity) and thus may need different configurations to be applied for the tuning parameters.

The pre-analysis also identifies the regions in the application that contribute significantly to the execution of an application and are referred to as *significant regions*. If no dynamism is identified in the pre-analysis, the rest of the READEX steps are aborted due to the homogeneous behaviour of the application, which will not yield any energy or performance savings from auto-tuning.

Listing 1.1 presents an example of the summary of significant regions and the dynamism identified by READEX in the miniMD application.

Listing 1.1: Summary of application pre-analysis

```
...
Significant regions are:

void Comm::borders(Atom&)
void ForceLJ::compute_halfneigh(Atom&, Neighbor&, int) [with int EVFLAG = 0; int
     GHOST_NEWTON = 1]
void ForceLJ::compute_halfneigh(Atom&, Neighbor&, int) [with int EVFLAG = 1; int
     GHOST_NEWTON = 1]
void Neighbor::build(Atom&)

Significant region information
==============================
Region name                Min(t)      Max(t)      Time Dev.(%Reg) Ops/L3miss Weight(%Phase)

void Comm::borders(Atom&)  0.001       0.001       2.6             109        0
void ForceLJ::compute_hal  0.013       0.014       2.9             97         68
```

```
void ForceLJ::compute_hal 0.016          0.016          0.0          91          1
void Neighbor::build(Atom 0.047          0.048          0.7          332         23

Phase information
=================
Min                Max                Mean               Dev.(% Phase)      Dyn.(% Phase)

0.0138626          0.0664566          0.020337           72.731             258.612

...

SUMMARY:
========

Inter-phase dynamism due to variation of execution time of phases

No intra-phase dynamism due to time variation

Intra-phase dynamism due to variation in compute intensity of following significant regions

void ForceLJ::compute_halfneigh(Atom&, Neighbor&, int) [with int EVFLAG = 0; int
     GHOST_NEWTON = 1]
void Neighbor::build(Atom&)
```

3.3 Derivation of Tuning Model

Following the identification of exploitable dynamism in the pre-analysis step, the third step explores the space of possible tuning configurations, and identifies the optimal configurations of the tuning parameters for the phases and rts's during the application execution. This analysis is performed by PTF (Periscope Tuning Framework) in conjunction with Score-P and the RRL (READEX Runtime Library). PTF performs DTA experiments through a number of possible search strategies, such as exhaustive, individual, and heuristic search based on generic algorithm to identify the optimal configurations for the rts's of the significant regions identified in the pre-analysis step. To achieve this, Score-P provides the instrumentation and profiling platform, while the RRL provides the platform for libraries to tune hardware and system parameters. Additionally, READEX also has dedicated libraries that tune application-specific parameters.

It is important to note that the additional identifiers, which can be specified during the instrumentation step provide additional domain knowledge to distinguish and identify different optimal configurations for runtime scenarios [5].

After all experiments are completed and optimal configurations are identified, the rts's are grouped into a limited number of scenarios, e.g., up to 20. Each scenario is associated with a common system configuration, and it is hence composed of rts's with identical or similar best configurations. The limitation in the number of scenarios inhibits a too frequent configuration switching at runtime that may result in higher overheads from auto-tuning. The set of scenarios, information about the rts's associated with the scenarios, and the optimal configurations for each scenario are stored by PTF in the form of a serialized text file called the Application Tuning Model (ATM), which is loaded and exploited during production runs at runtime.

The `readex_intraphase` plugin executes multiple tuning steps. First, the plugin executes the application for the default parameter settings. Next, it determines optimal configurations of the system level tuning parameters for different rts's. Finally, the plugin checks the variations in the results of the previous step and computes the energy savings. The `readex_intraphase` plugin performs an additional tuning step if application-level tuning parameters are specified for tuning, as described in Section 9.1.

4.1.1 Default Execution During this step, PTF executes the plugin with the default configuration of the tuning parameters to collect the program's static information after starting the application. The default settings are provided by the batch system for the system parameters. PTF uses a specific analysis strategy to gather program regions and rts's only for the first phase of the application. The measurement results are required to evaluate the objective value, for example, time and energy, and are used later to compare the results in the verification step.

4.1.2 System Parameter Tuning The system-level parameter tuning step investigates the optimal configuration for system-level tuning parameters. The plugin uses a user-defined search strategy to generate experiments. The search strategy is read from the configuration file. Exhaustive search generates a tuning space with the cross-product of the tuning parameters, and leads to the biggest number of configurations that are tested in subsequent program phases. The individual strategy reduces the number significantly by finding the optimum setting for the first tuning parameter, and then for the second, and so on. With the random strategy, the number of experiments can be specified. If no strategy is specified, the default individual search algorithm is selected. The experiments are created for the ranges of the tuning parameters as defined in the configuration file. The plugin evaluates the objective for the phase region as well as the rts's. The best system configuration is determined as the one that generates the least energy consumption. This knowledge is encapsulated in the ATM.

4.1.3 Verification The verification step is performed by executing three additional experiments in order to check for variations in the results produced in the previous step. For this purpose, PTF configures RRL with the best system configuration for the phase region and the rts-specific best configuration for the rts's. The RRL switches the system configurations dynamically for the rts's.

The plugin then determines the static and dynamic savings for the rts's and static savings for the whole phase before generating the ATM. The following three values characterize the savings at the end of the execution:

1. Static savings for the rts's: The total improvement in the objective value with the static best configuration of the rts's over the default configuration.
2. Dynamic savings for the rts's: The total improvement in the objective value for the rts's with their specific best configuration over the static best configuration.

with the maximum distance to the line formed by the points with the minimum and maximum average 3-NN distances.

The plugin then selects the best configuration for each cluster based on the normalized objective value. The cluster-best configuration represents one best configuration for all the phases of a particular cluster, and individual best configurations for the rts's in the cluster.

4.2.2 Default Execution The tuning plugin restarts the application and performs the same number of experiments as the previous step. In each experiment, the phase is executed with the default system configuration, i.e., the default settings provided by the batch system for the system tuning parameters. The plugin uses the objective values obtained for the phases and the rts's to compute the savings at the end of the tuning plugin.

why is the default execution here the second step? For intra-phase plugin it was 1. step ??

4.2.3 Verification The plugin restarts the application and executes the same number of experiments as the previous tuning steps. In each experiment, the plugin executes the phase with the corresponding cluster-best configuration and the rts's with their individual-best configurations. Phases that were considered noise in the clustering step are executed with the default configuration.

After executing all the experiments, the plugin modifies the Calling Context Graph (CCG) [10] by creating a new node for each cluster, and then clones the children of the phase region. It stores the tuning results and the ranges of the features for the cluster in each newly created node.

Like the `readex_intraphase` plugin, the `readex_interphase` plugin computes the savings as described in Section 4.1.3. However, the `readex_interphase` plugin aggregates the improvements across all the clusters.

5 Tuning Model Generation

At the end of the DTA, the Application Tuning Model (ATM) is generated based on the best system configuration computed via the tuning plugins. A best configuration is found for each rts. One option at runtime is to switch the configuration each time a new rts is encountered. However, this would result in very frequent switching, with a corresponding switching overhead (time and energy). To avoid this, READEX merges rts's into scenarios, and assigns a common configuration for all rts's in that scenario. The obvious solution is to merge rts's with identical best found configurations into scenarios. Since the set of possible configurations is very large, this can still result in too frequent switching. Hence, READEX merges rts's with similar configurations into scenarios and uses one common configuration for all rts's in a scenario.

The resulting ATM consists of the complete set of rts's, each determined by their identifier values and each with a link to the corresponding scenario. In addition, the ATM contains a list of the scenarios, each with a corresponding

[10] A context sensitive version of the call graph

of a region enter event from Score-P, and performs a check to detect if the encountered region is significant by searching for the region in the ATM. If the region is found, it is marked as a significant region. Otherwise, it is marked as an unknown region.

unclear how granularity is determined without creating at this point. - is this part of the ATM?

Another check is performed to determine if the region must be tuned by computing the granularity of the region. The region will be tuned only if the granularity is above a specified threshold, which defaults to 100 ms. Once the region is determined to be both significant and coarse-granular enough, additional identifiers that are used to identify the current rts are requested. The current rts can then be identified by both the call stack and the additional identifiers. Finally, a new configuration is applied for the current rts to perform the configuration switching. For a coarse-granular region that is marked as unknown, the calibration mechanism is invoked.

7.2 Configuration Switching

The scenario detection step applies a new configuration to the current region only if the region entered is found to be significant and coarse-granular. The setting for each tuning parameter is controlled by a dedicated Parameter Control Plugin (PCP). The RRL switches a configuration by sending a request to the PCPs.

The RRL supports two different modes for configuration switching: *reset* and *no-reset*. The *reset* mode maintains a configuration stack. Whenever a new configuration is set, the previous configuration is pushed onto this stack. When the corresponding unset occurs (e.g., if an instrumented region is left), the element is removed from the stack and the previous configuration is re-applied. If the *no-reset* mode is selected, the current configuration remains active until a new configuration is set, and the unset is ignored. This behaviour is configurable by the user. By default, the *reset* mode is enabled.

calibration may be wrong title here !?

7.3 Calibration

READEX makes a distinction between seen and unseen rts's. For known or seen rts's that are already present in the ATM, RRL simply uses the optimal configuration, and performs configuration switching. For the unseen rts's, the RRL calibration mechanism, described in Section 9.4 is used to find the optimal system configuration based on machine learning algorithms.

} extrapolation of unknown rts

8 Dynamic Switching Visualization

During DTA, PTF runs experiments with different configurations to find the optimal configuration for each rts in the application, which are then stored in the tuning model. During RAT, the RRL applies the optimal configuration from the tuning model for each rts during the application run. Hence, both stages require configuration switching during the application run.

tuning model from individual tuning models generated for program runs with different inputs, and is described in Section 9.3. Section 9.4 presents the calibration extension that covers runtime tuning for rts's that were not seen during design-time.

9.1 Application Tuning Parameters

Applications running on a cluster are aimed to solve numerical problems. Usually, the solutions can be computed with different methods, such as numerical integration (Simpson, Gaussian Quadrature, Newton-Cotes, ...), function minimization (Gradient Descent, Conjugate Gradient Descent, Newton, ...), or finding the eigenvalues and eigenvectors of real matrices (power method, inverse power method, Arnoldi, ...). These methods may have different implementations, like the Fast Fourier Transform (algorithms from Cooley-Tukey, Bruun, Rader, Bluestein) implemented in the FFTW, FFTS, FFTPACK or MKL.

For a given problem, several methods can provide similar numerical solutions through different implementations. In HPC, the developer chooses the most efficient one in terms of numerical accuracy and time to solution. However, the selection also depends on the computer's architecture. For example, some methods can be more efficient on a vector processor than on a superscalar processor. It is up to the application's developer to choose the appropriate method and its implementation that fulfill the computer's specificity.

In the context of energy saving, the energy consumption must also be minimized. This makes it hard for the developer to choose which method and implementation must be executed. READEX offers the developer to expose the various methods and their implementation via Application Tuning Parameters (ATP) to the tuning process. ATPs are communicated to READEX through an API that is used to annotate the source code at locations where the tuning parameters play a role.

Figure 4 illustrates the steps performed to tune the ATPs. First, ATPs are declared in the code through annotations. The application is then linked against the ATP library, which implements the API. During DTA, the ATP description file is generated by the ATP library. This file includes the tuning parameters with their specifications, and is used to define the search space. The `readex_intraphase` tuning plugin extends the search space with the ATPs, and finds the best settings. Finally, the best configurations are stored in the ATM and passed to RAT.

To annotate the code, the application developer exposes control variables and mark them as ATPs with the API functions as illustrated in Listing 1.2. The function `ATP_PARAM_DECLARE` declares the parameter's name, type, default value and domain. The function `ATP_PARAM_ADD_VALUES` allows to add possible values the parameter can take, in a range specified by minimal, maximal and increment values or by enumerating explicitly the possible values. The function `ATP_PARAM_GET` fetches the value given in the ATM from the RRL and assigns it to the control variable.

Several ATPs can be defined in the same code. They may be independent or not. In the latter case, there is a notion of constraints between the parameters.

the individual ATMs are discarded. Next, the `tuning model merger` filters all rts's in order to avoid duplicated rts's in the final tuning model. The next step is to produce a new set of scenarios by clustering all rts's as described in Section 5. Finally, the `tuning model merger` serializes the merged tuning model information and outputs the new ATM in the JSON format.

9.4 Runtime Calibration

As described in Section 3.4, RAT distinguishes known and unknown rts's. Known rts's are those which have been encountered during DTA. Here, the optimal configuration for the rts's is known. However, unknown rts's describe those which have not been encountered during DTA. There are several reasons why unseen rts's might occur. First, an unseen rts may consist of already known regions, but with other user parameters that were not present during DTA or parameters that might have changed between DTA and runtime. Typically, these changes are related to different application inputs. Second, an unseen rts may consist of completely new regions, which were not seen during DTA. The goal of the calibration is to handle these unknown rts's during RAT.

have limits

Since the calibration is done during the production run, there are a few restrictions which had to be taken in account for the design of the calibration mechanism. First, there can be no user input and second, a good configuration has to be found in a short time. A DTA like approach to search for an optimal configuration is not feasible as it would degrade the performance of the application.

For runtime calibration, READEX provides two different machine learning based approaches. The first approach is a Q-Learning based mechanism. The algorithm starts at a given core and uncore frequency. With a probability of ϵ, the algorithm selects a different setting from the next direct neighbours. Then, it measures the energy at this point. Based on the measurement, it calculates the so-called Q-Value. Afterwards, the algorithm chooses the next optimal status according to the Q-Value and starts from the beginning. Figure 5 shows how the mechanism searches for an energy efficient optimum (core frequency = 2.3 GHz, uncore frequency = 2.2 GHz) starting from a selected initial setting (core frequency = 1.9 GHz, uncore frequency = 2.2 GHz).

why performance counters here? is this related to the need for more inputs to the shallow neural network - in comparison to a deep network ?

In the second approach, the RRL uses performance counters to predict the optimal setting using neural networks. First, the approach identifies the most relevant performance counters. In a second step, it trains a shallow neural network in order to predict a good frequency. This model is then loaded during runtime. Once the RRL encounters an unseen RTS, it measures the relevant performance counters and uses the neural network to predict a good setting. In contrast to the Q-Learning approach, this approach does require a pre-training, which has to be performed once per system. However, once this is done, the network just needs to be evaluated, which requires less tuning and runtime overhead compared to the Q-Learning approach.

For the memory bound solver (GMRES), manual tuning resulted in a low core frequency, high uncore frequency and the use of eight threads to overcome Non-Uniform Memory Access (NUMA) effects of the dual socket computational node.

	assemble_k [J]/[s]	assemble_v [J]/[s]	gmres_solve [J]/[s]	print_vtu [J]/[s]	main [J]/[s]
default setings	1467/5.4	1484/ 5.9	2733/10.2	1142/5.6	6872/27.3
static tuning	1962/9.8	2015/10.6	1366/ 6.1	420/2.4	5792/29.0
dynamic tuning	1476/7.0	1462/ 7.2	1259/ 7.9	293/2.1	4531/24.3
static savings [%]	-33.8/-82.3	-35.8/-79.1	50.0/40.5	63.2/56.8	15.7/-6.2
dynamic savings [%]	-0.6/-30.6	1.5/-20.9	53.9/23.2	74.3/62.9	34.0/10.9

Table 2: Comparison of the energy and time consumption for the default, optimal static, and dynamic configurations of BEM4I.

The energy and time consumption of each region in the application in optimum static and optimum dynamic (optimum configuration for the region itself) configuration is presented in Table 2. While static savings reached 15.7 %, the dynamic switching among individual configurations increased the savings to 34.0 % on the Haswell nodes. Decrease in the run time in this case was caused by NUMA effects of the MKL solver – the tuned version runs on eight threads, and due to the compact affinity, all threads run on a single socket. It can also be seen that the optimum static configuration has a very bad impact on the `assemble_k` and `assemble_v` regions, and also results in a suboptimal behavior of the region `print_vtu`.

is this a detail for the table caption? here it is bit disturbing the flow of reading

10.2 Application Parameters tuning

As mentioned in Section 9, READEX comes with two approaches to tune application parameters: (1) using the ATP library, and (2) using Application Configuration Parameters (ACP). The integration of the ATP library requires developer knowledge of the application and therefore, we implemented this support into the ESPRESO library, which was developed by IT4Innovations in the READEX project.

A long list of ATPs were evaluated: runtime tuning of FETI METHOD (2 options), PRECONDITIONERS (5 options), ITERATIVE SOLVERS (2 options), HFETI type (2 options), SCALING (2 options), BO_TYPE (2 options), NON-UNIFORM PARTS (6 options), REDUNDANT LAGRANGE (2 options) and adaptive precision (2 options). For runtime tuning of domain decomposition (10 options), the developer had to implement the support for this parameter, since ESPRESO performs domain decomposition only during startup. For READEX, we developed an enhanced ESPRESO to redo the decomposition after each time-