European Commission
Horizon 2020
European Union funding
for Research & Innovation

GA no. 671657

READEX
Runtime Exploitation of Application Dynamism
for Energy-efficient eXascale computing

D4.1
Concepts for the READEX Tool Suite

Document type:	Report
Dissemination level:	Public
Work package:	WP4
Editor:	Michael Lysaght (ICHEC-NUI Galway)
Contributing partners:	N/A
Reviewer:	Per Gunnar Kjeldsberg (NTNU)
	Joseph Schuchart (TUD)
Version:	0.2

Document history

Version	Date	Author/Editor	Description
0.1	01/02/16	Michael Lysaght, Kashif Iqbal (ICHEC-NUIG), Joseph Schuchart, Andreas Gocht (TUD), Michael Gerndt, Anamika Chowdhury, Madhura Kumaraswamy (TUM), Per Gunnar Kjeldsberg, Magnus Jahre, Mohammed Sourouri (NTNU), David Horák, Lubomír Říha, Radim Sojka, Jakub Kruzik (IT4I-VSB), Kai Diethelm (GNS), Othman Bouizi (Intel)	1st Draft
0.2	15/02/16	Michael Lysaght, Kashif Iqbal (ICHEC-NUIG), Joseph Schuchart, Andreas Gocht (TUD), Michael Gerndt, Anamika Chowdhury, Madhura Kumaraswamy (TUM), Per Gunnar Kjeldsberg, Magnus Jahre, Mohammed Sourouri (NTNU), David Horák, Lubomír Říha, Radim Sojka, Jakub Kruzik (IT4I-VSB), Kai Diethelm (GNS), Othman Bouizi (Intel)	2nd Draft
1.0	26/02/16	Michael Lysaght, Kashif Iqbal (ICHEC-NUIG), Joseph Schuchart, Andreas Gocht (TUD), Michael Gerndt, Anamika Chowdhury, Madhura Kumaraswamy (TUM), Per Gunnar Kjeldsberg, Magnus Jahre, Mohammed Sourouri (NTNU), David Horák, Lubomír Říha, Radim Sojka, Jakub Kruzik (IT4I-VSB), Kai Diethelm (GNS), Othman Bouizi (Intel)	Final Draft

Executive Summary

The objective of Work Package 4 (WP4) ‘READEX Tool Suite Development’ is to integrate the developed READEX techniques and software components into the overall READEX tool suite. As such, one of the early objectives of WP4 is to specify the concepts and formalism of the overall tool suite, as well as to define the interfaces between the design-time and runtime components.

In this deliverable, we describe the fundamental concepts of the READEX methodology. By presenting a formal mathematical description of the READEX concepts for the first time, one of the central aims of the deliverable is to minimise ambiguity in their definitions, to improve common understanding and communication between the project partners, as well as to support the development of READEX methodologies and software components throughout the lifetime of the project.

In READEX, we distinguish between three different levels of the High Performance Computing (HPC) stack, i.e., hardware, runtime system, and application-level. Multiple parameters across the HPC stack have so far been identified as being relevant to READEX. We continuously emphasise throughout the deliverable that all of these tuning parameters can be influenced at runtime, a characteristic that is fundamental to the READEX approach. When describing the READEX methodology, we expand on how the tuning approach can be supported by domain knowledge, specified by the READEX tool suite user.

We describe how the READEX tool suite will leverage two existing software tools, namely the Periscope Tuning Framework (PTF) and the Score-P instrumentation and measurement infrastructure. The READEX project will see the development of a new READEX Run-time Library (RRL). In addition to a high-level description of how this software infrastructure will serve the READEX tool suite, we provide an architectural description of the software components, including the software extensions that will be implemented during the READEX project. We also provide a description and release plan for READEX tool suite prototypes, as well as a plan to ensure software quality throughout the development phases of the project.

Finally, we describe a case study that centres around a widely used HPC application in the engineering domain, which will serve to guide the design of the READEX tool suite throughout the project.

The deliverable reflects the content of the following internal Working Documents (WDs): WD1.1 ‘Description of Available Tuning Parameters’, WD2.1 ‘Description of the Concepts for Scenario Identification and Configuration Pre-Computation’ and WD3.1 ‘Description of the READEX Runtime Library Architecture.’ With the finalisation of this document, milestone MS1 ’System parameters for exploiting dynamism have been established’ is also completed. While this deliverable and corresponding WDs reflect the current READEX concepts as well as the overall approach and architecture, it should be noted that these may be refined over the course of the project.

Contents

1 Introduction

As part of the US Department of Energy (DOE)'s Exascale Computing Initiative [?], both the US DOE and US National Nuclear Security Administration (NNSA) have set out the goal to develop and deploy a productive Exascale system with a power envelope of 20 MW by 2023, which is just above the power dissipation of today's leading systems. This goal is also reflected by the ETP4HPC Strategic Research Agenda [?]. As a result of the 20 MW power constraint, the High Performance Computing (HPC) community (as well as the data centre market) has been experiencing a shift from merely focusing on the maximum *performance* of an application running on a HPC system, to focusing also on its *performance per watt*, to prepare for the Exascale era.

Several measures that influence the energy consumed when running a software application on an extreme-scale HPC system are available to developers, including hardware settings, system software parameters, and application characteristics. However, developers typically focus on implementing and optimising algorithms for accuracy and performance and neglect possible improvements to the energy-efficiency of the application running on the HPC system. The fact that developers typically lack the platform and hardware knowledge, as well as tools, required to influence the energy consumption means that improvements to the energy efficiency of applications have, to date, been rarely targeted.

With this challenge in mind, the objective of the READEX project is to deliver the first stand-alone auto-tuning framework that has the capability to automatically and dynamically tune a wide breadth of large-scale HPC applications at design- and run-time as we progress from deep-Petascale to Exascale computing. In developing such a tools-aided auto-tuning methodology, the project aims to enable developers to achieve significant improvements in the energy-efficiency of current and future applications on extreme-scale systems, while at the same time significantly increasing productivity relative to manual tuning.

The auto-tuning concept, which combines application performance analysis and tuning based on the trial-and-error tuning of a set of predefined strategies, has been investigated and employed for more than two decades on a wide range of computing systems [?]. Existing auto-tuning frameworks explore optimal configurations at design-time and assume a fixed selection of the best configurations throughout the complete production run of an application (static auto-tuning). This assumption will no longer hold for Exascale systems. Instead, an application will also be required to adapt to both resource changes and its own behaviour during production runs (dynamic auto-tuning). While a small number of dynamic auto-tuning methodologies and prototype tools exist that target dynamic optimisations during production runs [?, ?], no single standalone dynamic auto-tuning framework currently exists with the capability to target the full breadth of extreme-scale HPC applications being exploited in academia and industry, both now and on the road to Exascale.

A key technology objective of READEX is to focus on targeting extreme-scale applications and, as such, to develop an auto-tuning framework that can scale to extreme node counts. Inspired by the system scenario based design methodology [?, ?, ?] used in the embedded systems community, READEX will develop the concept of (semi)-distributed dynamic tuning, which aims to minimise centralisation of control, which is known to impede scalability due to global synchronisations. One of the central aims of READEX is to instead perform synchronisations as locally as possible (e.g., restrained to a node and only if required by the tuning parameters employed), thereby avoiding scaling bottlenecks. With the system scenarios methodology at its core, the READEX framework will be a lightweight infrastructure, which will result in minimal overhead in monitoring and tuning at run-time.

The majority of current auto-tuning frameworks and optimisation methodologies focus on single objective optimisations, be it tuning for either time or energy. One of the major challenges on the road to Exascale is the ability to tune an application for energy, but at the same time ensure that the application's performance does not suffer as a consequence. Therefore, the READEX tool suite will be specifically designed for efficient multi-objective tuning for time and energy based on pre-computing Pareto curves.

One of the most important opportunities for improving an application's energy-efficiency is based on the fact that HPC applications have characteristics that vary during their execution. This variation, called *application dynamism*, is what READEX exploits to improve energy efficiency. Typical characteristics of application dynamism include application compute intensity (compute/memory bound), load balancing (balanced/unbalanced), algorithm granularity (coarse/fine), as well as parallel efficiency (high/low).

In exploiting application dynamism, READEX will target applications that exhibit iterative behaviour, typically in the form of a main progress loop that, for example, iterates over time steps during a simulation. These individual time steps are known as *phases* in the execution of an application. Applications that contain such phases exhibit phase dynamism, which can be further extended to the concept of intra- and inter-phase dynamism. The *intra-phase dynamism* results from the execution of different algorithms within a single phase, whereas the *inter-phase dynamism* results from changing characteristics of the algorithms while the simulation is progressing through the sequence of phases.

Fundamental to the READEX approach is the exploitation of this intra- and inter-phase dynamism to guide in the optimal tuning of the HPC stack. In READEX, the term HPC stack covers the hardware, the runtime system, and the application layer. We will investigate parameters on all of these levels, which we refer to generically as the set of *tuning parameters*. Included in this deliverable is a wide-ranging early-stage analysis of the tuning parameters, which we plan to target as part of the READEX project.

The central technical aim of the READEX project is to combine the concept of application dynamism with the automatic energy and performance tuning of HPC stack parameters that are tunable at runtime. As described in detail in this deliverable, this approach will span two major parts of the HPC application life-cycle, namely, (1) application development and performance tuning at design-time, and (2) performance tuning during the runtime of applications in production mode. In the READEX methodology, we refer to these two parts of the life-cycle as *Design Time Analysis* (DTA) and *Runtime Application Tuning* (RAT), respectively, as shown in Figure **??**.

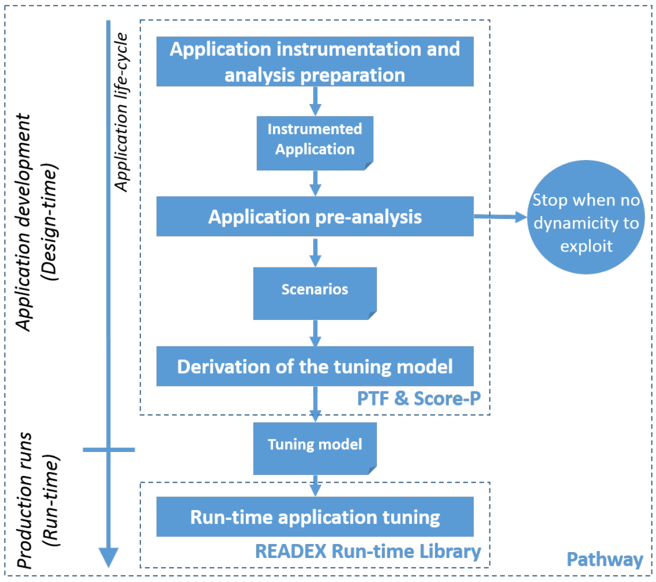

Figure 1: High-level view of READEX tools-aided methodology

During DTA, analysis of the dynamism exhibited by a target HPC application is performed. This analysis will be made possible by an extended version of Periscope Tuning Framework (PTF) [?] to be further developed during the READEX project. PTF was developed as part of the EU FP7 ICT AutoTune project [?] and is an extension of Periscope [?], an automatic distributed performance analysis tool developed by TUM, towards a generic platform supporting various automatic tuners for static performance and energy-efficiency optimisation. PTF's main principles are the use of formalised expert knowledge and strategies complemented by the automatic execution of experiments. Its design is based on an extensible and modular architecture that exploits *tuning plugins* and allows for distributed, scalable processing. PTF already provides a number of predefined tuning plugins as well as a tuning plugin interface for the development of new plugins and where individual tuning plugins can be combined into so-called *meta-plugins*. As described in detail in this deliverable, PTF will be used as a core component of the DTA software infrastructure, where it will be significantly extended with the capabilities for the detection of application dynamism as well as for the search for optimal values for relevant HPC stack tuning parameters, which we refer to as *system configurations*.

The search for optimal system configurations is based on the evaluation of an *objective*, such as the minimisation of energy consumption or the minimisation of time-to-solution. Such measurements will be carried out by PTF during DTA via the Score-P instrumentation and measurement infrastructure [?]. Through its flexible design, Score-P offers different strategies for application instrumentation, including support for compiler-based and manual region instrumentation, which both insert calls to event handlers into the application code.

Moreover, Score-P also supports several parallelisation paradigms such as the Message Passing Interface (MPI, [?]) and OpenMP [?], all of which will be employed in READEX. The different instrumentation points are triggered upon events that are caused by the application during execution and are handled by event handlers inside the Score-P monitoring library. These events are used for various means of data processing and storage inside Score-P, including trace and profile generation as well as an online access interface for direct analysis at runtime, as used by PTF.

As detailed in this deliverable, all of the knowledge obtained during DTA is encapsulated in a *tuning model*. This tuning model captures the knowledge about the best-found system configurations for individual scenarios. It is based on the idea of *scenario*-based tuning, which aggregates similar *runtime situations* (rts's) into scenarios, where an rts could be a particular function call on a given process during a given loop iteration. This partitioning of the set of rts's into scenarios is based on the sharing of common best-found system configurations. In its most basic form, the tuning model will contain a look-up table providing the best-found system configurations for the known scenarios determined during DTA. This tuning model is subsequently forwarded to a low-overhead library, which we refer to as the *READEX Runtime Library* (RRL). The aim of the RRL is to ensure that the system is configured in an optimised way for the execution of each rts.

During production runs, the READEX methodology uses the previously obtained knowledge about application dynamism and the best-found system configurations to adapt the HPC stack parameters to the changing characteristics of the target application. This task is carried out by the RRL linked to the application. While running the application in production mode, the RRL is called on every entry to instrumented code regions and where the forthcoming scenario is predicted using a classifier created during DTA. In most cases, the result of the prediction will be an already known scenario, in which case the look-up table from the tuning model is employed to provide the best-found system configurations. The RRL will include a switching mechanism, which will tune the HPC stack tuning parameters defined by the system configurations via extensions being developed for the Score-P infrastructure. As a result of successful switching, the application will continue its execution in the best-found system configuration.

The remainder of this deliverable expands on the READEX concepts, methodology and framework as described so far. In Section 2, we present the fundamental concepts and formalism of the READEX methodology. In Section 3, we describe the tuning parameters that we will initially target as part of the READEX project. In Section 4, we describe the overall READEX tool suite approach, followed in Section 5 by a description of the tool suite from an integrated architecture perspective. Finally, in Section 6 we provide an application case study that has acted to guide our concepts for the project so far.

2 READEX Concepts and Formalism

The goal of this section is to present a formal mathematical description of the READEX concepts to minimize ambiguity in their definitions, to improve common understanding and communication between the project partners, and to support the development of READEX DTA and RAT methodologies and software components.

In Section **??**, we present the fundamental concepts of the READEX methodology. Section **??** presents the concept of a tuning model which is the final outcome of the DTA stage and which is passed to the RRL to guide dynamic runtime tuning of the HPC stack.

In Section **??**, we describe how the base READEX concepts are applied to define the *tuning potential* metric, a metric quantifying the potential improvement to be achieved by the tuning, and thus central to the overall READEX methodology. Section **??** extends the base concepts to include the central concept of inter-phase dynamism. Finally, in Section **??** we describe how these fundamental concepts can be extended to handle a multiplicity of *application inputs*, which also typically leads to varying application behaviour during runtime.

2.1 Fundamental concepts

This section introduces the core concepts that will be referred to throughout this document and the READEX project more generally.

2.1.1 Significant regions

The set of all instrumented regions in the program, e.g., functions, parallel OpenMP regions, and MPI operations, is denoted by R_{instr}, i.e., all regions that are known to the READEX tool suite. During execution of the application, a region might be executed multiple times. Each region execution is called a *region instance.* It is assumed that for all regions $r \in R_{instr}$ the instrumentation overhead is insignificant.

A region $r \in R_{instr}$ is called a *significant region* if it covers a significant part of the execution time. The READEX tuning approach only targets the significant regions. The set of significant regions are denoted $R_{sig} \subseteq R_{instr}$. From this point in the deliverable, the subscript *sig* may be omitted, in which case R stands for R_{sig}.

2.1.2 Identifiers

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

The set of all identifiers is denoted as ID. $ID_r \subseteq ID$ is the set of identifiers for a region $r \in R$ and ID_R is the set of identifiers of all regions. Identifiers of regions considered in READEX include the *region name*, *region call path*, and *region parameters.* The region call path is the sequence of nested region instances when a region is executed. Region parameters are variables associated with the region as identifiers (see Section **??** for more details).

Further subsets of the set ID are specified in Section **??** and Section **??**. Values that can be taken by identifiers are denoted as $v \in V$.

2.1.3 Context elements

A *context element* c is a tuple consisting of an identifier and its value. $C := ID \times V$ is the set of all context elements, and is defined as the cross product of all identifiers and the values taken by them.

The set of context elements for region $r \in R$ is defined as $C_r := \{c \in C | c = (id, v) \wedge id \in ID_r\}$.

2.1.4 Runtime situations

A *runtime situation* (rts) is an instance of a significant region $r \in R$ during an execution. Instances of regions $r \in R_{instr} \setminus R_{sig}$ are not rts's. The set of all possible runtime situations is denoted by RTS.

The region of an $rts \in RTS$ is denoted by $r_{rts} \in R$ and its context by $C_{rts} \subseteq C$. Multiple rts's may have the same context.

2.1.5 Application execution

An application is executed by a set of *processes* P, each of which can use multiple threads. An *execution* of the application on a given input by a process p is $exe_p := rts_1, rts_2, \ldots, rts_n$, where $n = len(exe)$. An *application execution* is the set of executions of all processes and is defined as $EXE := \{exe_p | \forall p \in P\}$.

2.1.6 Tuning parameters

A *tuning parameter* $tp \in TP$ is a parameter of the HPC stack (e.g. CPU frequency, accelerator offloading switch, application parameter, etc.). READEX focuses on tuning parameters that have the potential to influence the energy consumption of an application running on an extreme-scale system and can be affected by the RRL at runtime.

A tuning parameter $tp \in TP$ can take a value from VAL_{tp}. The set of all values of tuning parameters is $VAL = \cup_{tp \in TP} VAL_{tp}$. See Section **??** for possible tuning parameters considered in READEX.

2.1.7 System configurations

A *system configuration* $cfg \in CFG$ is a function that maps a tuning parameter $tp \in TP$ onto its value $val \in VAL_{tp}$ and is defined as $cfg : TP \longrightarrow VAL$.

2.1.8 Switching point

A *switching point* $sp \in SP$ is a point during application execution when the monitoring library is entered and can perform switching of the system configuration. The monitoring library can determine the region r at every switching point. Switching points are triggered when a region is entered or exited.

The set of sequences of switching points during a given execution exe_p is defined as $SP_{exe_p} := \{sp_p | p \in P \wedge sp_p = sp_1^p, sp_2^p, \ldots, sp_n^p \wedge n = len(sp_p)\}$.

2.1.9 Objective function

The *objective function* $o : RTS \times CFG \longrightarrow \mathbb{R}$ is a function mapping a given runtime situation rts and a given system configuration onto a real number. The *objective* of tuning is to minimise or maximise a given objective function by varying the system configuration.

2.1.10 Static system configuration

In READEX, we consider a *static system configuration* $cfg_{static} \in CFG$ as a static system configuration that is either a system-wide default or was obtained by tuning for a best static configuration. For example, we could select the best fixed CPU frequency for an entire program run, optimising the energy consumption of the application.

The static system configuration cfg_{static} gives a baseline for a comparison of the results achieved with READEX runtime tuning.

2.1.11 Example

This section describes the terms introduced in Section **??** with an example code.

The program in Listing **??** consists of the `main()` function that calls three other functions, namely `laplace()`, `reduction()`, and `fftw()`. After instrumenting the application with Score-P, we obtain $R_{instr} = \{laplace, reduction, fftw\}$, the set of all regions in the application. In READEX, the most interesting regions are the ones that are coarse-granular so that there is low overhead while switching between configurations (see Section **??**).

Suppose the execution times of each instance of `laplace()`, `reduction()`, and `fftw()` are 2 s, 0.1 ms and 1 s respectively. The execution times for `laplace()` and `fftw()` are much

Listing 1: Code example

```c
int main(void) {

   // Initialize application
   // Initialize experiment variables

   int num_iterations = 2;
   for (int iter = 1; iter <= num_iterations; iter++) {
      laplace();              // significant region
      residue = reduction(); // insignificant region
      fftw();                 // significant region
   }

   // Post-processing:
   // Write noise matrices to disk for visualization
   // Terminate application

   MPI_Finalize();
   return 0;
}
```

greater than the switching overhead. Thus, `laplace` and `fftw` are significant regions $R_{sig} = R = \{laplace, fftw\}$ and will be the targets of the READEX tuning approach.

The region name and call-path of these functions can be selected as identifiers. For example, identifiers for region `laplace` would be $ID_{laplace} = \{RegionName, CallPath\}$.

Suppose we have two processes, P_0 and P_1, executing this program. An rts of region `laplace` in process P_1 for iteration $iter = 1$ of the loop is denoted as $rts^{1,P_1}_{laplace}$. Assuming that we have two iterations, all possible runtime situations are denoted by

$$RTS = \{rts^{1,P_0}_{laplace}, rts^{1,P_1}_{laplace}, rts^{2,P_0}_{laplace}, rts^{2,P_1}_{laplace}, rts^{1,P_0}_{fftw}, rts^{1,P_1}_{fftw}, rts^{2,P_0}_{fftw}, rts^{2,P_1}_{fftw}\}.$$

A context element $c_{rts^{1,P_0}_{laplace}} = (CallPath, main/laplace)$ for region `laplace` is a tuple containing the identifier $CallPath$ and its value. $C_{rts^{1,P_0}_{laplace}} = \{(RegionName, laplace), (CallPath, main/laplace)\}$ is the set of all context elements of this rts. The application execution on process P_0 is defined as

$$exe_{P_0} = rts^{1,P_0}_{laplace}, rts^{1,P_0}_{fftw}, rts^{2,P_0}_{laplace}, rts^{2,P_0}_{fftw}.$$

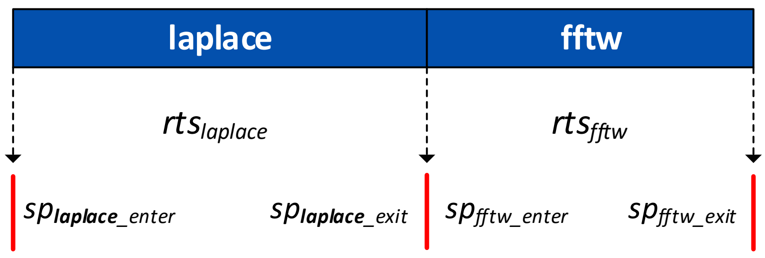

Figure 2.1: Sequence of switching points and runtime situations

Switching points are triggered when a region is entered or exited and $SP = \{sp_{laplace\_enter}, sp_{laplace\_exit}, sp_{fftw\_enter}, sp_{fftw\_exit}\}$ is the set of switching points. The sequence of switching points and rts's for process P_0 are shown in Figure **??**.

The goal of READEX is to optimize an objective function $o(rts, cfg)$ by switching the system configuration dynamically at the switching points. In this example, the tuning parameter $tp \in TP$ that influences the energy consumption is CPU_Freq, the CPU frequency setting (See Section **??**). CPU_Freq can take values from $VAL_{CPU\_Freq} = \{1.8, 2.0, 2.1, 2.3, 2.4, 2.5\}$.

The system configurations

$$
\begin{aligned}
cfg_0(CPU\_Freq) &= 1.8\\
cfg_1(CPU\_Freq) &= 2.0\\
cfg_2(CPU\_Freq) &= 2.1\\
cfg_3(CPU\_Freq) &= 2.3\\
cfg_4(CPU\_Freq) &= 2.4\\
cfg_5(CPU\_Freq) &= 2.5
\end{aligned}
$$

map CPU_Freq to different values. Let the system-wide static configuration be $cfg_{static}(CPU\_Freq) = 2.5$. The application execution on process P_0 results in Table **??** of measured energy values.

2.2 Tuning model

As a result of DTA, a tuning model is produced. The tuning model captures the knowledge about the best-found system configurations for individual rts's. It is based on the idea of scenario-based tuning which aggregates similar rts's into scenarios.

Table 1: Energy measurement values for process P_0 in Joules

cfg	CPU_Freq	$rts_{laplace}$		rts_{fftw}	
		$o(cfg, rts^{1,P_0}_{laplace})$	$o(cfg, rts^{2,P_0}_{laplace})$	$o(cfg, rts^{1,P_0}_{fftw})$	$o(cfg, rts^{2,P_0}_{fftw})$
0	1.8	1	4	3	4
1	2.0	2	5	4	1
2	2.1	3	2	2	3
3	2.3	6	8	1	2
4	2.4	11	13	7	6
5	2.5	14	15	10	12

2.2.1 Scenarios

The set of rts's is partitioned into a set, S, of scenarios. The rts's are grouped into one scenario if they have the same best-found configuration, i.e., the same selector as defined in Section **??** below, or if they have the same context C_{rts}.

2.2.2 Classifier

A *classifier* $cl : \mathcal{P}(C_R) \longrightarrow S$ maps each $rts \in RTS$ onto a unique scenario $s \in S$ based on the rts context C_{rts}.

2.2.3 Selectors

A *selector* of a scenario $s \in S$ is a function $sel_s : \emptyset \longrightarrow CFG$ that returns a single configuration. The configuration is usually the best configuration with respect to the chosen objective. The set of all selectors is SEL. Selectors can have quite different implementations based on the knowledge included in the tuning model for the scenario. Examples of selectors are:

- It is based on a single best configuration determined at design time.
- It is based on a set of pareto-optimal configurations and returns one of those according to some runtime priorities for the objectives.
- It could choose from a set of good configurations determined during design time analysis based on probabilities. This could actually dynamically check for the best solution and go beyond the limitation of design time analysis.
- It could select from different configurations that were combined because of merging rts's with different best configurations, but with the same context (see Section **??** for an example).

Selectors are also used to store the tuning information for individual rts's, which we call *rts selectors*.

2.2.4 Example (continued)

This section describes the terms introduced in Section **??** using the example defined in Section **??**.

Table **??** contains the measured energy values for the objective function $o(rts, cfg)$. The values of the consumed energy for $o(cfg, rts^{1,P_0}_{laplace})$ and $o(cfg, rts^{2,P_0}_{laplace})$, and for $o(cfg, rts^{1,P_0}_{fftw})$ and $o(cfg, rts^{2,P_0}_{fftw})$ vary for configurations cfg_0 to cfg_5.

The classifier $cl(C_{rts})$ groups $rts^{1,P_0}_{laplace}$, $rts^{2,P_0}_{laplace}$, rts^{1,P_0}_{fftw} and rts^{2,P_0}_{fftw} into scenarios $s \in S$ if they have the same context or if they have the same selector. Since $rts^{1,P_0}_{laplace}$ and $rts^{2,P_0}_{laplace}$ have the same context

$$C_{rts^{1,P_0}_{laplace}} = C_{rts^{2,P_0}_{laplace}} = \{(RegionName, laplace), (CallPath, main/laplace)\}$$

they are grouped into one scenario. Similarly, rts^{1,P_0}_{fftw} and rts^{2,P_0}_{fftw} have the same context

$$C_{rts^{1,P_0}_{fftw}} = C_{rts^{2,P_0}_{fftw}} = \{(RegionName, fftw), (CallPath, main/fftw)\}$$

and are grouped into another scenario. Thus, we have two scenarios:

$$s_0 = \{rts^{1,P_0}_{laplace}, rts^{2,P_0}_{laplace}\}$$
$$s_1 = \{rts^{1,P_0}_{fftw}, rts^{2,P_0}_{fftw}\}$$

It may be argued that the rts's of `laplace` and `fftw` have different best configurations, and so, they should be partitioned into different scenarios. This does not happen because it is not possible to differentiate between the rts's based on the context, and hence, they are merged into the same scenario. These cases will happen rarely if the identifiers are well chosen.

The selector returns a single best configuration from configurations that were combined due to the merging of $rts^{1,P_0}_{laplace}$ and $rts^{2,P_0}_{laplace}$ into s_0 and rts^{1,P_0}_{fftw} and rts^{2,P_0}_{fftw} into s_1. The selector $sel_{s_0}() = cfg_2$ chooses the configuration $cfg_2(CPU\_Freq) = 2.1$ for scenario s_0 and $sel_{s_1}() = cfg_3$ chooses $cfg_3(CPU\_Freq) = 2.3$ for scenario s_1.

2.3 Applying the formalism

This section describes how the concepts described so far can be applied, both in terms of *tuning-relevant dynamism* and in terms of the tuning potential.

Table 2: Optimal system configurations for rts's

$ocfg(rts^{1,P_0}_{laplace}, o)$	$ocfg(rts^{2,P_0}_{laplace}, o)$	$ocfg(rts^{1,P_0}_{fftw}, o)$	$ocfg(rts^{2,P_0}_{fftw}, o)$
cfg_0	cfg_2	cfg_3	cfg_1

2.3.1 Optimal system configuration

The function $ocfg : RTS \times OBJ \longrightarrow CFG$ is an oracle and determines the optimal system configuration for a runtime situation and a given objective. It is the goal of READEX to approximate *ocfg* as best as possible based on the READEX methodology.

2.3.2 Tuning-relevant dynamism

An application exhibits *tuning relevant dynamism* iff $\exists rts, rts' \in RTS$ such that $ocfg(rts, o) \neq ocfg(rts', o)$. This means that there are two rts's during one execution $exe_p \in EXE$ that have different optimal configurations with respect to objective o. In such cases, the system configuration has to be switched in order to achieve the best possible objective function value.

2.3.3 Tuning potential

The *tuning potential* $tpo : RTS \times OBJ \longrightarrow \mathbb{R}$ is a function that maps a runtime situation onto a real value specifying the improvement in the objective function compared to a static configuration cfg_{static}. It is defined as $tpo(rts, o) := o(rts, ocfg(rts, o)) - o(rts, cfg_{static})$.

The tuning potential for the execution $exe_p \in EXE$ is an extension of *tpo* for sequences of rts's. It is defined as $tpo(exe_p, o) := \sum_{n=1}^{len(exe_p)} tpo(rts_n, o)$. The tuning potential of the entire application run is defined as $tpo(EXE, o) := \sum_{p \in P} tpo(exe_p, o)$.

Whether the tuning potential can be achieved with the READEX methodology depends on the quality of the tuning model, the switching overhead, and mutual side effects between processes.

2.3.4 Example (continued)

This section extends the example defined in Section **??** and Section **??** to describe the terms introduced in Section **??**.

The optimal system configuration function $ocfg(rts, o)$ returns an optimal configuration $cfg \in CFG$ for each rts as shown in Table **??**.

During the application execution, we see that $rts^{1,P_0}_{laplace}$ and $rts^{2,P_0}_{laplace}$ have different optimal configurations with respect to the objective. Hence, tuning relevant dynamism exists because a configuration switch is determined to be worthwhile.

The tuning potential $tpo(rts^{1,P_0}_{laplace}, o) = o(rts^{1,P_0}_{laplace}, ocfg(rts^{1,P_0}_{laplace}, o)) - o(rts^{1,P_0}_{laplace}, cfg_{static})$ for $rts^{1,P_0}_{laplace}$ specifies the improvement in the value of the objective function (consumed energy) as compared to the value for $cfg_{static}(CPU\_Freq) = 2.5$. The improvement for $rts^{1,P_0}_{laplace}$ is 92.857%, and for the application execution exe_{P_0} is 90.196%. However, the tuning effect is diminished because of the merging of the rts's. Due to this limitation, we can only realize an improvement of 88% for execution exe_{P_0}. To avoid this, an additional identifier can be introduced to distinguish the rts's (see Section **??**).

2.4 Extension for inter-phase dynamism

This section describes an extension of the formalism for inter-phase dynamism. We define an inter-phase tuning model, which is a tuning model according to the definition in Section **??** and can provide different system configuration for rts's in phases with different characteristics. The definitions in this section and in Section **??** align strongly with the definitions for identifiers, scenarios, and tuning model in Section **??**, which demonstrates the generality of the fundamental concepts.

2.4.1 Phase region

A *phase region* is a program region $r \in R_{instr}$ that defines the phases of execution. All significant regions should be included in the dynamic extent[1] of the phase region. The set of all phase regions is $R_{phase} \subseteq R_{instr}$. In READEX, we initially assume that there is exactly one phase region, i.e., $|R_{phase}| = 1$.

2.4.2 Phase

A *phase* $ph \in PH$ is an instance of a phase region and thus an rts on phase level. It is a single execution of the phase region. In READEX we assume that the phases are executed collectively by all of the processes. Thus, all the processes go through the same *phase sequence* $ps = ph_1, ph_2, \ldots, ph_k$.

2.4.3 Phase identifiers

For the phase region $r \in R_{phase}$, identifiers can be given, as for any other significant region. We denote the *set of phase identifiers* as $ID_{phase} \subseteq ID$. The phase identifiers should be chosen such that they allow to distinguish different behaviour across phases of that phase

[1]The dynamic extent of a region is all the code executed during a region instance. This is different from the code textually nested in the region.

region. For region identifiers, we define the context elements based on phase identifiers as $C_{phase} \subseteq C$.

2.4.4 Inter-phase tuning model

Based on the phase identifiers, the rts's of a region with the same context but in phases with different characteristics can now be mapped to different rts scenarios with different selectors. Thus, we obtain again a set of scenarios, S, which might be different from the one for the same execution ignoring phases. The partitioning into rts scenarios is now given by a *phase-aware classifier* $cl : \mathcal{P}(C_{phase} \cup C_R) \longrightarrow S$.

Since the tuning model, $TM = (S, cl, SEL)$, is based on a phase-aware classifier, it can now cover the scope of inter-phase dynamism .

2.4.5 Example (continued)

This section describes the terms introduced in Section **??** with an extension to the example code described in Section **??**.

The program in Listing **??** now consists of the phase region $r \in R_{phase}$. ph_1 is the first execution of r, the body of the for loop and the first loop iteration. Phase ph_2 is the second loop iteration. Thus, the sequence of phases is $ps = ph_1, ph_2$. All the processes execute the same sequence.

In Section **??**, both the rts's of `laplace` and `fftw` were merged into scenarios s_0 and s_1 respectively, and the tuning potential was diminished. To avoid this, a phase identifier is introduced, $ID_{phase} = \{PhaseCharact\}$, which distinguishes phases with different characteristics. Let $PhaseCharact = A$ for ph_1 and $PhaseCharact = B$ for ph_2. Now, the merged rts's can be distinguished because we can draw information about a phase's characteristic, and thus determine which phase an rts belongs to.

Listing 2: Code example with phase region

```c
int main(void) {

   // Initialize application
   // Initialize experiment variables

   int num_iterations = 2;
   for (int iter = 1; iter <= num_iterations; iter++) {
      // Start phase region
      // Read PhaseCharct
      laplace3D();             // significant region
      residue = reduction(); // insignificant region
      fftw_execute();          // significant region
      // End phase region
   }

   // Post-processing:
   // Write noise matrices to disk for visualization
   // Terminate application

   MPI_Finalize();
   return 0;
}
```

The phase contexts $c_{ph_1} = (PhaseCharact, A)$ and $c_{ph_2} = (PhaseCharact, B)$ form the set of all phase contexts $C_{phase} = \{(PhaseCharact, A), (PhaseCharact, B)\}$. The contexts of the rts's of `laplace` and `fftw` are now extended to include the phase contexts as

$$\begin{aligned} C_{rts^{1,P_0}_{laplace}} =& \{(RegionName, laplace), (CallPath, main/laplace), \\ & (PhaseCharact, A)\} \end{aligned}$$

$$\begin{aligned} C_{rts^{2,P_0}_{laplace}} =& \{(RegionName, laplace), (CallPath, main/laplace), \\ & (PhaseCharact, B)\} \end{aligned}$$

$$\begin{aligned} C_{rts^{1,P_0}_{fftw}} =& \{(RegionName, fftw), (CallPath, main/fftw), \\ & (PhaseCharact, A)\} \end{aligned}$$

$$\begin{aligned} C_{rts^{2,P_0}_{fftw}} =& \{(RegionName, fftw), (CallPath, main/fftw), \\ & (PhaseCharact, B)\} \end{aligned}$$

The phase-aware classifier $cl(C_{rts}) \longrightarrow s$ where $s \in S$ returns the following scenarios:

$$\begin{aligned}
s_0 &= \{rts^{1,P_0}_{laplace}\}\\
s_1 &= \{rts^{1,P_0}_{fftw}\}\\
s_2 &= \{rts^{2,P_0}_{laplace}\}\\
s_3 &= \{rts^{2,P_0}_{fftw}\}
\end{aligned}$$

The selectors for the scenarios return the following configurations:

$$\begin{aligned}
sel_{s_0} &= cfg_0\\
sel_{s_1} &= cfg_3\\
sel_{s_2} &= cfg_2\\
sel_{s_3} &= cfg_1
\end{aligned}$$

The scenarios, the phase-aware classifier and the selectors form the inter-phase tuning model $TM = (\{s_0, s_1, s_2, s_3\}, cl, \{sel_{s_0}, sel_{s_1}, sel_{s_2}, sel_{s_3}\})$. With this tuning model, the full tuning potential for P_0 can be realized.

2.5 Extension for multiple application inputs

This section presents an extension of the previous concepts for different application runs with different application inputs. The application input comprises input data and execution environment configurations. Application runs with different inputs will usually result in different performance and energy characteristics. Since in READEX, we assume that the application has reproducible performance and energy characteristics when run with the same input several times, the tuning model should be extended to account for this varying behaviour.

2.5.1 Application input

The *application input* $ai \in AI$ includes the input data to the application as well as other application-external settings at the start of an application that influence the performance and dynamic behaviour of the execution, such as the set of processes P. As a result of the application behaving differently for different application inputs, the DTA will have to consider a set of application inputs. AI denotes all possible application inputs for the given application.

2.5.2 Input identifier

Identifiers can be given for the application input that can be used to differentiate different performance and energy characteristics. Such identifiers are called *input identifiers* and are denoted by ID_{input}. The context elements for those identifiers are denoted by C_{input}.

2.5.3 Application tuning model

Based on the application input identifiers we can now obtain the application tuning model that considers application input, application phases, and significant regions. It is based on a partitioning of all rts's in all executions for the different inputs into rts scenarios. The partitioning will be given by an input-aware classifier $cl : \mathcal{P}(C_{input} \cup C_{phase} \cup C_R) \longrightarrow S$ that maps rts's based on the input context, the phase context, and the rts context into rts scenarios.

2.5.4 Example (continued)

This section describes the terms introduced in Section **??** with reference to the code in Listing **??**.

Different application inputs with different characteristics $i_1, i_2, i_3, \ldots$ result in different measured values of the consumed energy. Assuming there is an input identifier $InputCharact$, which takes the values $i_1, i_2, i_3, \ldots$ and identifies these characteristics, it can be used to differentiate different energy characteristics.

The contexts of the rts's of `laplace` and `fftw` can now also contain the contexts of the input identifiers. The contexts of the rts's for the first execution are:

$$\begin{aligned} C_{rts^{1,P_0}_{laplace}} =& \{(RegionName, laplace), (CallPath, main/laplace), \\ & (PhaseCharact, A), (InputCharact, i1)\} \end{aligned}$$

$$\begin{aligned} C_{rts^{2,P_0}_{laplace}} =& \{(RegionName, laplace), (CallPath, main/laplace), \\ & (PhaseCharact, B), (InputCharact, i1)\} \end{aligned}$$

$$\begin{aligned} C_{rts^{1,P_0}_{fftw}} =& \{(RegionName, fftw), (CallPath, main/fftw), \\ & (PhaseCharact, A), (InputCharact, i1)\} \end{aligned}$$

$$\begin{aligned} C_{rts^{2,P_0}_{fftw}} =& \{(RegionName, fftw), (CallPath, main/fftw), \\ & (PhaseCharact, B), (InputCharact, i1)\} \end{aligned}$$

The contexts of the rts's for the second execution are:

$$\begin{aligned}C_{rts_{laplace}^{1,P_0}} =&\{(RegionName, laplace), (CallPath, main/laplace),\\ &(PhaseCharact, A), (InputCharact, i2)\}\end{aligned}$$

$$\begin{aligned}C_{rts_{laplace}^{2,P_0}} =&\{(RegionName, laplace), (CallPath, main/laplace),\\ &(PhaseCharact, B), (InputCharact, i2)\}\end{aligned}$$

$$\begin{aligned}C_{rts_{fftw}^{1,P_0}} =&\{(RegionName, fftw), (CallPath, main/fftw),\\ &(PhaseCharact, A), (InputCharact, i2)\}\end{aligned}$$

$$\begin{aligned}C_{rts_{fftw}^{2,P_0}} =&\{(RegionName, fftw), (CallPath, main/fftw),\\ &(PhaseCharact, B), (InputCharact, i2)\}\end{aligned}$$

The application tuning model will now contain the new rts scenarios, the input-aware classifier cl_{TM} and the selectors for the new rts scenarios.

2.6 Summary

The previous sections presented a formalism to describe the READEX methodology as well as the tool suite described in detail in later sections. The tuning model serves as the interface between the DTA and RAT stages and is based on a partitioning of rts's into scenarios. For each scenario, the decision logic for selecting the system configuration to use is provided by a selector, which in turn returns a system configuration. The rts's are mapped by a classifier to those scenarios. In the tuning model, the classifier is based on the input context, the phase context, and the context of the rts. As a result, all of the sources of application dynamism can be used to select a system configuration.

3 Tuning Parameters

The concepts described in the previous chapter will be used to determine the optimal parameter settings for running a given rts. In our description of READEX-relevant tuning parameters, we distinguish between three different levels of the HPC stack, i.e., hardware parameters, system software parameters, and application-level parameters. The parameters that we have so far identified as being potentially relevant to the READEX project are summarised in Table **??** and will be described in more detail below. It should be emphasised that all of the parameters in Table **??** can be tuned at runtime, which is a vital property for dynamic auto-tuning. Any parameter that can only be statically changed before application start-up will not be considered in the READEX project.

While the parameters in Table **??** are considered relevant to READEX, further investigations are required to confirm that the impact they have on the energy-efficiency of applications in production is significant enough to warrant inclusion in the READEX tool suite. Finally, parameters that share a similar impact on energy efficiency or parameters that require coordinated control have been combined into so-called *tuning aspects*, where such tuning aspects will need to be handled in a coordinated way to avoid contravening settings.

We will initially focus on the the Taurus system installed at TU Dresden.[2] This system is comprised of 1456 nodes with each hosting containing two 12-core Intel Xeon CPUs E5-2680 v3 (Intel Haswell processor family) running with a default frequency of 2.50 GHz [**?**]. The nodes contain between 64 and 256 GB of memory. Additionally, 44 nodes are equipped with two Nvidia Tesla K20x GPUs and 64 nodes contain four Nvidia Tesla K80 GPUs per node.

3.1 Hardware parameters

Currently, the most relevant hardware settings that we consider for investigation are processor related parameters. This is due mainly to the fact that the processor has the highest power dissipation in a computer system. In the following sections, we will describe each of the hardware tuning parameters and tuning aspects that we will investigate as part of the READEX project.

3.1.1 Processor core frequency

The method of Dynamic Voltage and Frequency Scaling (DVFS) has been investigated since the 1990's [**?**] as a means of reducing the energy consumption of computer systems. Reducing the frequency of a CPU core through DVFS results in the reduction of the required power draw of the platform, where energy savings of up to 32% have been reported [**?**]. DVFS can be implemented through various means, e.g., by changing the governor, which is a pluggable infrastructure that commonly controls the CPU frequency settings in the Linux operating system based on defined policies such as *performance*, *powersave*, or *ondemand.*

[2] `https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/SystemTaurus`

Table 3: Overview on envisioned tuning parameters

Level	Tuning Aspect	Tuning Parameter	Scope
Hardware Parameters	CPU Frequency Controls	Frequency Scaling (DVFS)	Core
		Duty Cycle Management (DDCM)	Core
		Uncore frequency	Socket
		Energy Performance Bias (EPB)	Socket
	Memory Prefetcher	Prefetcher Setting	Socket
System Software Parameters	MPI Reduction	Short message size threshold	Application
		SMP-awareness	Application
		SMP message size threshold	Application
	MPI Alltoall	Short message size threshold	Application
		Medium message size threshold	Application
	OpenMP Threading	Dynamic Concurrency Throttling	Process
		Workload scheduling algorithm	Process
Application Parameters	User-specified code-paths	Decomposition	Application
		Compiler type & compiler parameter	Application
		Type of iterative & direct solvers	Application
		Preprocessing of stiffness & coarse problem matrices	Application
	Dynamic offloading	OpenMP target device	Process

For full control over the frequency settings, the so-called `userspace` governor allows the arbitrary applications to select the so-called P-state, which are essentially frequency steps of the processor.

One interesting point to emphasise is that the Intel Haswell processor family allows for the independent selection of P-states for individual cores as opposed to full sockets as seen in previous processor lines. The Intel Haswell processor also introduces a switching window for frequency changes that causes a delay of up to 500 μs before switching decisions go into effect [?].

3.1.2 Processor uncore frequency

Transfers to and from memory are controlled by the so-called uncore frequency. The Haswell processor usually sets this frequency independently of the CPU core frequencies. However, it is also possible to control the uncore frequency from userspace by setting machine specific registers (MSRs) to control the uncore frequency. For this purpose, a library will be installed on the machine Taurus that allows controlled unprivileged access to MSRs [?].

In the READEX project, we will investigate the potential for energy savings by reducing the uncore frequency for compute intensive regions and subsequently increasing it for memory-bound regions. Since the uncore and core power domains share the same thermal budget, we will investigate potential performance benefits to memory operations when reducing the core frequency, thus shifting power budget to the uncore of the processor and potentially increasing memory performance [?].

3.1.3 Dynamic Duty Cycle Modulation (DDCM)

A further possibility to reduce power consumption on a processor is by taking advantage of a feature known as Dynamic Duty Cycle Modulation (DDCM). This technique involves so-called T-states which instruct the processor to statistically skip a user-defined number of clock cycles, i.e., between 12.5 % and 87.5 % of the overall clock cycles for a give time period. This can be beneficial in program regions in which not all cycles can be used effectively, e.g., memory- or I/O-bound regions or wait-states. By skipping idle cycles, energy is saved through clock-gating. DDCM is expected to suffer less from the delay introduced by the frequency switching window of the Haswell processor family described in Section **??**. Energy savings up to 20% have been reported under controlled circumstances, e.g., for rebalancing the execution of otherwise unbalanced OpenMP parallel regions [?].

3.1.4 Energy Performance Bias (EPB)

The Energy Performance Bias (EPB) is a setting introduced with the Intel Haswell processor family that controls different energy efficiency related features on the processor, e.g., the turbo mode [?]. The processor monitors the current execution to automatically adapt to the current compute requirements. The processor offers three settings: *performance*, *energy saving*, and *balanced*, providing a coarse-grain control over the features of the processor. We expect that changing the EPB can change parameters on the CPU that are not accessible through software. As such, we will investigate how achievable energy savings compare to the manual tuning of parameters such as CPU core frequency.

3.1.5 Hardware Prefetcher

Hardware prefetchers aim to predict upcoming memory access patterns based on the recent access pattern. Enabling them can lead to performance improvements for regular strided accesses. Unfortunately, random memory accesses are not predictable, leading to a high number of unnecessary memory loads consuming bandwidth that becomes unavailable to the application. Disabling the prefetchers can therefore lead to a higher bandwidth of random memory access pattern [?]. With higher memory performance being available to the application, the performance may increase, thus leading to improved energy-efficiency.

3.2 System software parameters

The majority of scalable HPC applications employ the Message Passing Interface (MPI) for distributed memory parallelism, with many HPC applications now also employing OpenMP to target thread-level parallelism as part of a so-called hybrid model. Due to their widespread adoption within the HPC community, both of these programming models will be investigated throughout the READEX project for runtime tuning opportunities.

3.2.1 Message Passing Interface (MPI)

The MPI 3.0 standard has introduced the so-called MPI-T interface that allows MPI implementations to offer a set of parameters that can be read and written by the user. These variables can be used to optimise an MPI implementation for a specific environment, or in the case of the READEX project to adjust the implementation to the needs of a given application during runtime. We will initially focus our investigations on the latest version of the well-known open source MPICH implementation [?] of MPI, which offers the largest and most relevant set of parameters for runtime tuning. For example, MPICH exposes the definition of short and long messages for many MPI commands as well as parameters influencing shared memory operations. Unfortunately, most MPI parameters need to be changed in a synchronised fashion throughout the whole MPI environment, i.e., on all processes, which is expected to pose extra challenges for efficient tuning during runtime.

In Table **??**, we point to several MPI tuning parameters that we consider relevant to the READEX project and that we will initially investigate. The parameters shown represent so-called collective communication operations. Relative to direct point-to-point communication between two MPI processes, collective operations are more expensive, as more processes are involved. For example, `MPI_Alltoall` communicates data from all processes to all processes and therefore is among the most expensive MPI commands in terms of communication. `MPI_Reduce` collects results from different given processes and can carry out different calculations, e.g. it is possible to determine the minimum of a value about all processes. Therefore, the call is not only communication intensive, but also compute intensive. We will tune different parameters related to these MPI commands and investigate their tuning potential.

3.2.2 OpenMP

The OpenMP standard offers users a means to implement thread-parallelism through preprocessor statements, which are translated by the compiler into thread-parallel code. For example, one of the most commonly used OpenMP parallelisation techniques is offered by the OpenMP `parallel-for` directive, in which all iterations of a loop are distributed among a defined number of threads. At the same time, the OpenMP standard provides an API to control the behaviour of the OpenMP runtime library, e.g., to influence the number of threads used by a parallel region and to control workload scheduling policies, as summarised in Table **??**.

Table 4: MPI tuning parameters to be further investigated.

Tuning Aspect	Tuning Parameter	Description
MPI Reduce	Reduction Short Message Size	The short message algorithm will be used if the send buffer size is ≤ this value (in bytes)
	SMP-awareness	Enable SMP aware reduce
	SMP Message Size Threshold	Maximum message size for which SMP-aware reduce is used. A value of '0' uses SMP-aware reduce for all message sizes
MPI Alltoall	Alltoall Short Message Size	The short message algorithm will be used if the per-destination message size (`sendcount*size(sendtype)`) is ≤ this value
	Alltoall Medium Message Size	The medium message algorithm will be used if the per-destination message size (sendcount*size(sendtype)) is ≤ this value and larger than the short message size

Table 5: OpenMP tuning parameters to be further investigated.

Tuning Aspect	Tuning Parameter	Description
OpenMP Threading	Number of Threads	The number of threads can be controlled via `omp_set_num_threads()` to perform Dynamic Concurrency Throttling (DCT).
	Load scheduling algorithm	Dynamically change the load scheduling algorithm of thread-parallel loops. Requires `OMP_SCHEDULE` be set to `dynamic`.

The default number of OpenMP threads equals the number of logical CPUs in the system or any value enforced via the environment variable `OMP_NUM_THREADS`. Reducing the number of threads allows unused processor cores to reduce their C-state, i.e., to go from the active C0 state into any of the power-saving states C1–C6 in which no computation can be performed [?, ?]. Thus, reducing the number of threads and letting a subset of the available cores go into a sleep-state can improve energy-efficiency, e.g., if the number of loop iterations is too small for all active threads or if the loop iterations are memory-bound and the memory bandwidth can be saturated by a smaller number of threads. Changing the workload scheduler policy can also be used to optimise the execution efficiency by controlling the distribution of work among the threads, which can have a positive impact if not all loop iterations require the same amount computational work.

3.3 Application parameters

As well as the hardware and system software parameters, the READEX project expects that the targeted application itself offers a certain degree of tuning potential through application-level parameters. So far, we have identified several potential parameters that can be exploited by automatically switching between possible settings as listed in Table **??**.

Parameters such as decomposition, solver and preprocessing model switches are highly application and problem specific and thus can only be handled generically through the definition of variables that are controllable by the tuning environment. Since the tuning library does not have any knowledge of the algorithmic background of the parameter space, the user has to specify the range of values that are suitable for the parameter under investigation. Additional specification of domain knowledge will be presented in Section **??**. Defining the interfaces for specifying the application-level tuning parameters will be a major task to be carried out in the second half of the first project year.

We have identified three different kinds of application-level tuning parameters, i.e., simple control variable parameters, alternative code paths, and dynamic offloading. For simple control variables, it is sufficient to define a variable that can be controlled by the tuning library and its possible values. For alternative code paths, the application implementation has to honour the value of a tuning variable to control the program flow. A special case of different code paths is the choice between offloading models for heterogeneous accelerators, such as GPUs or the Intel Xeon Phi coprocessor. Here, the choice does not require application-level background but an estimate of whether the data transfer and the execution on an accelerator might yield higher efficiency than the execution on the host.

3.4 Coordination of switching decisions

The scope of several of the parameters listed in Table **??** exceeds the boundary of a single process, e.g., the set of MPI parameters. These parameters have to be set to the same value on all processes of the parallel application run, which requires some form of coordination to prevent differing settings among the processes.

Different approaches can be taken to perform coordinated switching of parameters with a global scope. If the decision of which value to be set for a global parameter is dependent on information that is similar on all processes, no explicit communication is required. This can be the case for MPI parameters if the choice is made based on the number of processes or the number of Bytes to be transferred. This information has to be the same on all processes participating in a global MPI operation to ensure standard compliance.

If the decision making is not based on implicitly synchronised information, some sort of synchronisation has to be employed for parameters with a global scope. The first approach is to synchronise all participating processes immediately before the switching decision is being taken. This approach offers great flexibility since it allows different parameter settings to be taken for every switching decision throughout the application execution, i.e., at every switching point relevant for the respective parameter. However, synchronising processes with a high frequency will incur a significant overhead. Nevertheless, this approach is applicable if the number of relevant switching points is low.

Another approach is the synchronisation at the beginning or end of a phase region and a modification of the tuning model for the current or next phase based on the outcome of the coordination. This reduces the flexibility of global parameter choices to a per-phase granularity but significantly reduces the coordination overhead. This step can be implemented as part of the calibration mechanism introduced in Section **??**.

Table 6: Application-level tuning parameters to be further investigated.

Tuning Aspect	Tuning Parameter	Description
Control values	Decomposition parameter	Decomposition of a problem of fixed size into various number of subdomains significantly affects the behavioural of the FETI Solver in terms of iterative solver processing time, solver convergence, preprocessing time and memory usage
	Loop optimization factors	Factors controlling loop layout to adapt to the underlying hardware characteristics, e.g., cache sizes.
Alternative code paths	Type of iterative and direct solvers	Different direct solvers provide various efficiency for various input datasets. Iterative solver reduces preprocessing, while increasing the iteration time.
	Compiler type and compiler parameter	Compiler type (e.g. icc, gcc, llvm), optimization parameters (O0-O3).
	Preprocessing of stiffness and coarse problem (CP) matrices	Stiffness matrix (K) regularization and factorization; Schur complement – uses dense representation; sparse direct solver – uses sparse representation; CP assembling; CP matrix factorization, CP matrix explicit inverse. Various approaches to assemble and solve CP provides workload balancing between preprocessing and projector processing per iteration.
Dynamic offloading	OpenMP target device	By setting the OpenMP target device for `target` regions, the runtime library can choose between devices to offload, including whether the computations are executed on the host or offloaded to an accelerator.

4 READEX Tool Suite: The Approach

This section presents the READEX tool suite approach, which implements the concepts described in Section **??**. The approach for the DTA are described in Section **??** followed by the RAT stage in Section **??**. The concepts for the specification of domain knowledge are presented in Section **??**.

4.1 Design Time Analysis

Design Time Analysis (DTA) forms the first stage in the READEX methodology. In this stage, the targeted application is analysed and a tuning model is produced, which is stored by DTA and used by the RRL to guide dynamic tuning. The DTA stage leverages PTF and the Score-P measurement infrastructure. Specifically, DTA uses Score-P in combination with the RRL to instrument the target application, to set tuning parameters via tuning actions, and to measure application performance and energy consumption. The tuning model captures knowledge about intra-phase and inter-phase dynamism as well as about the characteristics of different application inputs. In this section, we first provide an overview of DTA, with subsequent sections providing more details on the individual steps involved during the DTA stage.

4.1.1 DTA overview

In Figure **??** we show the overall workflow of the DTA stage, which begins with the specification of the objective. The next step, the specification of the domain knowledge (Section **??**) via Score-P annotations is optional but typically involves the specification of at least a phase region R_{phase} of interest. The step is followed by the instrumentation of the targeted application via Score-P, where judicious instrumentation will typically be required to reduce instrumentation overhead (Section **??**). Subsequently, the significant regions R_{sig} of the application are detected (Section **??**), where such regions will be required to be coarse enough to result in a measurable impact from tuning. These regions will also be required to be nested in the phase region, but will not be mutually nested in early prototypes of the READEX tool suite.

Once the significant regions are identified, the tuning potential of the READEX methodology is estimated (Section **??**), where this estimation is based on the amount of application dynamism. If this analysis indicates that it is worthwhile applying the READEX methodology, the DTA stage progresses with the building of the tuning model.

Prior to starting the inter-phase analysis step, hints are first generated that guide the subsequent analysis process (Section **??**). Based on the hints, the inter-phase analysis step is initiated, where a sequence of tuning cycles (TC) is performed. In each cycle, a *phase tuning model* is determined for the current phase. At the end of the inter-phase analysis step an inter-phase tuning model is provided (Section **??**). The intra-phase analysis that is initiated in a TC leverages the PTF tuning plugins to determine a best system configuration for the

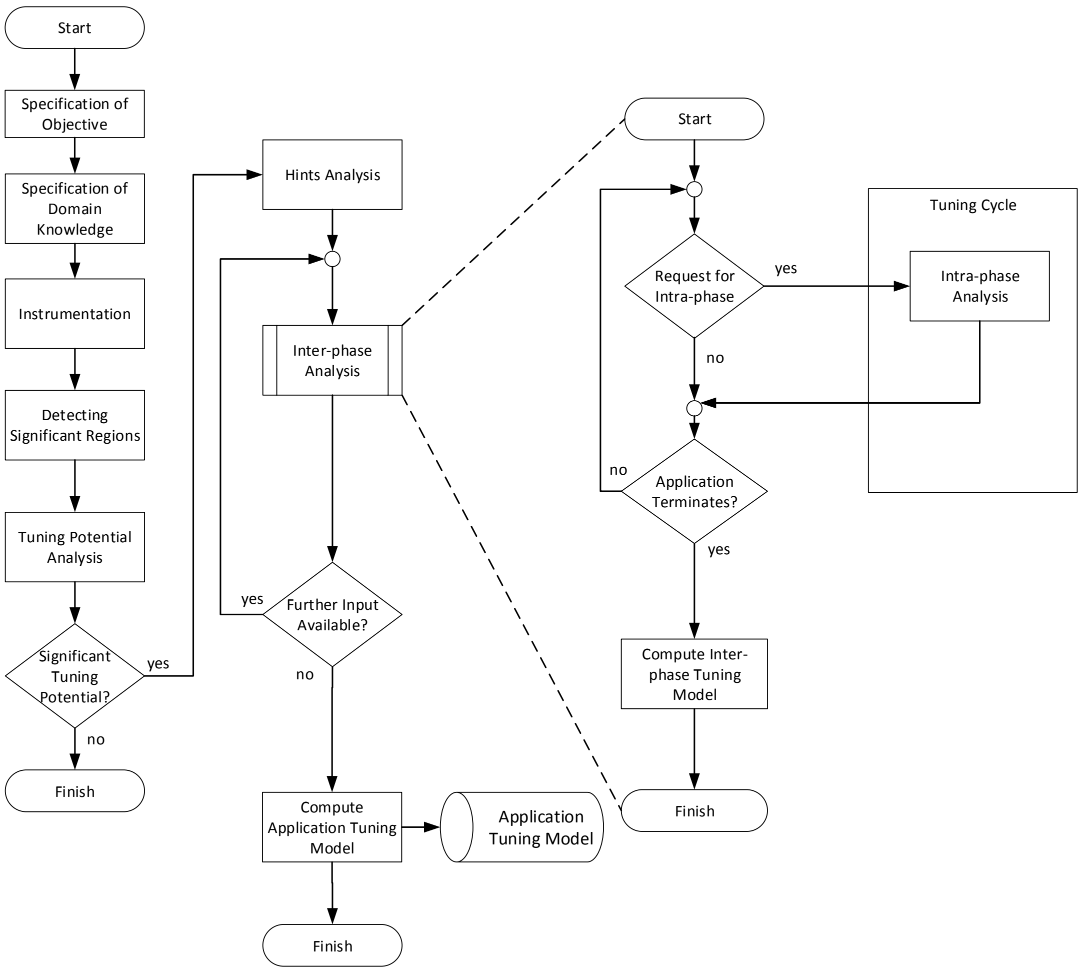

Figure 4.1: Design Time Analysis Workflow

significant regions (Section **??**). If the user has selected different application inputs for DTA, the analysis is repeated for the different application inputs and a final application tuning model is computed (Section **??**).

4.1.2 Instrumentation

The first step of the DTA after the optional specification of domain knowledge (Section **??**) is to instrument the application with Score-P. The instrumentation enables the measurement of program regions and dynamic runtime tuning with the RRL via calls to event handler in Score-P that are inserted at the enter and exits points of regions. Score-P can instrument various region types, such as application functions, MPI library calls, and OpenMP parallel regions. The instrumentation can be carried out either automatically or manually by the user.

While automatic instrumentation may be preferable for a user, this approach may lead to high overhead, which is typically caused by the instrumentation of frequently executed fine granular regions. To reduce this overhead, the Score-P monitor can be configured with a list of regions that should be omitted from measurement (even though they may have been instrumented). To select the regions that should not be measured, a threshold will be used for the granularity of the region, where the metric for granularity is determined by the average execution time of the instances of a region, expressed as

$$granularity = \frac{t_{incl}^{region}}{\#Instances}$$

Only if the *granularity* of a region is larger than this threshold, will the region be considered for the next steps of DTA. This threshold will be in the millisecond range so that the overhead for switching the configuration will be lower than the gains due to tuned execution. After the instrumentation, all regions that are not filtered during the execution of the application form R_{inst}. In the next DTA step, the significant regions R_{sig} are selected from this set. The optimisation of the instrumentation will require two runs of the application for determining the overhead and verifying successful filtering.

4.1.3 Detection of significant regions

After the instrumentation step, all remaining regions are coarse granular enough to be candidates for the READEX tuning methodology. In this step, the set of significant regions R_{sig} is selected, which are then considered for runtime tuning. In READEX, we will first consider tuning only for regions that are not dynamically nested. This means that none of the significant regions is executed as part of an instance of another significant region. An algorithm will be developed to select those significant regions from all instrumented regions. The detection of the significant region can be done based on the result of the second application run done in the previous step.

4.1.4 Predicting tuning potential

The goal of this step is to estimate the tuning potential of the application, where the outcome of this analysis is a Boolean value indicating whether it is worthwhile to progress to a more detailed analysis. The analysis will check whether there are variations in behaviour of the rts's of the significant regions. The application will be executed in this step and several metrics will be measured for each of the rts's. An important metric for variation is the execution time of the rts's, where differences in the execution time of rts's for a significant region indicate different characteristics. As well as execution time, other metrics might be considered, such as the compute intensity[3] of an rts, which is an indicator of whether the rts is memory or compute bound. If the metrics can be measured in a single run with the available hardware counters, this step might require only a single additional run of the application.

4.1.5 Intra-phase analysis

Intra-phase analysis will search for the best configuration for different rts's of significant regions within a given phase and will automatically select and execute READEX tuning plugins for the tuning aspects given in Section **??**. Tuning plugins optimise the application for a specific tuning aspect with certain tuning parameters. A plugin captures expert knowledge on how to search for the best configuration of those parameters. A plugin may use performance analysis information to reduce the search space, e.g., the range of values for the tuning parameters, and then apply a search strategy to evaluate different configurations via experiments and measuring the objective value.

Figure **??** shows the detailed steps during intra-phase analysis. This overall step executes the tuning plugins. Each plugin will determine a best configuration of its tuning parameters for an rts. The best configurations determined by different plugins will be combined into a global best configuration. Based on this best configuration the rts's will be partitioned into scenarios and the phase tuning model will be determined. The output of this step is the phase tuning model, as described in Section **??**.

4.1.6 Inter-phase analysis

The goal of the inter-phase analysis is to exploit dynamism across phases. During DTA, the intra-phase analysis will be periodically restarted, where each restart is a tuning cycle. The frequency of repeating intra-phase analysis can be determined by a PTF command line argument or be a result of the hint analysis executed as one of the first steps of DTA (See Section **??**). As well as starting a new TC with a fixed frequency, TCs can also be triggered by a user-provided-signal such as an event triggered by a grid adaptation in the application.

In Figure **??**, there are a total of 120 phases. The overall execution is sampled into m tuning cycles as TC_1, TC_2, ... TC_m. For each TC the intra-phase analysis is repeated and a phase tuning model is computed. Phase identifiers ID_{phase} have to be provided during the start

[3]Number of floating point operations per byte accessed in memory.

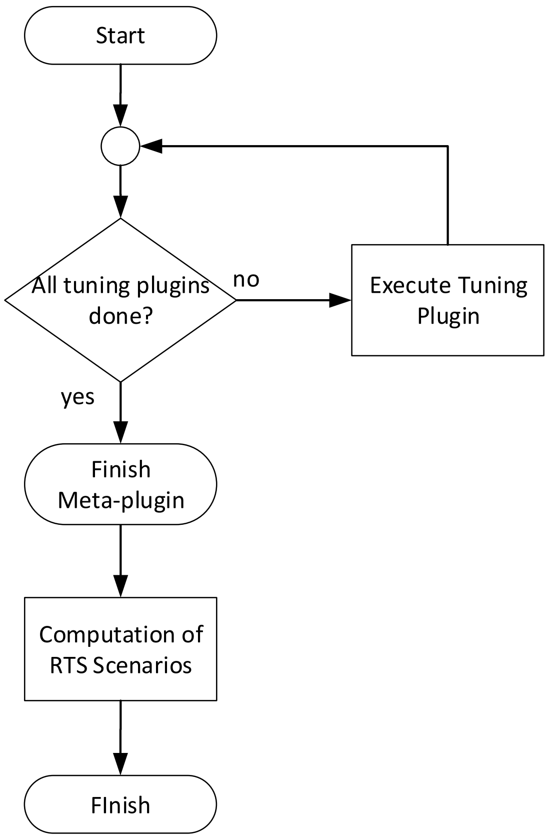

Figure 4.2: Intra-phase analysis workflow

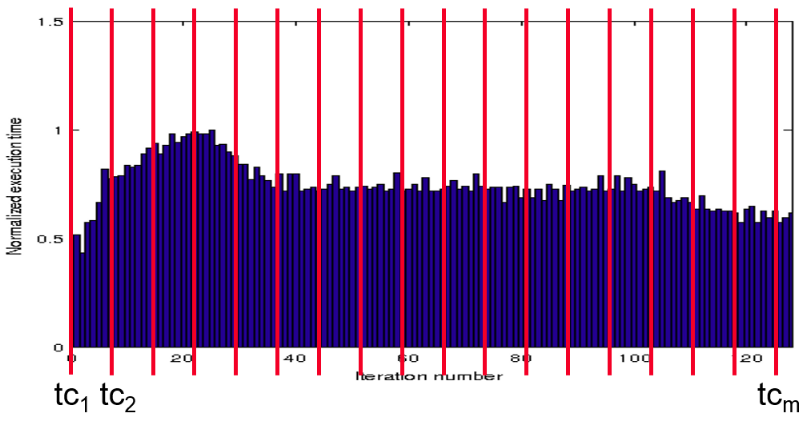

Figure 4.3: Sampling the overall signal of the Indeed application.

of the application as domain knowledge. The phase identifiers enable PTF to distinguish phases according to their behaviour and to generate more precise scenarios with a phase-aware classifier for the inter-phase tuning model introduced in Section **??**. A phase-aware classifier, cl_{phase}, uses the phase context in addition to the rts context to partition rts's into scenarios with similar characteristics. If no phase identifiers are provided, rts's in different phases cannot be distinguished, where even if rts's have different selectors in their phase tuning model, they will be merged into a single scenario in the tuning model. The output of this step is the inter-phase tuning model (Section **??**).

Intra- and inter-phase analysis will be done in a single program run.

4.1.7 Hints analysis

The hints analysis step will be executed before inter-phase tuning is initiated. The goal of this analysis is to automatically determine parameters influencing the approach taken in inter- and intra-phase analysis. One parameter is the frequency of tuning cycles: If changes in the phase characteristics happen with a certain frequency, e.g., every 100 phases, the inter-phase analysis can be configured to repeat the intra-phase analysis with this frequency. An additional parameter is the repetition of experiments to evaluate certain configurations during intra-phase analysis, which depends on the overall noise in the signal of the execution time of phases.

Figure **??** shows the individual execution times for 120 phases of an application. A certain variation can be seen in the execution time, which means that the measurements should be repeated for a few iterations to get a stable value. This step will output the settings for the inter-phase and intra-phase analysis parameters.

4.1.8 Tuning model generation

The goal of this step is to determine the application tuning model, which is then stored for later use be the RRL. If multiple application inputs are provided by the user, the DTA stage will cycle through multiple application executions with different inputs. This will result in input-specific inter-phase tuning models, which will then be combined into the tuning model. To distinguish different rts's for different inputs, application input identifiers are required, capturing different application characteristics. The final tuning model will now be based on an *input-aware classifier* mapping rts's to scenarios based on the input context, phase context and region context. The output of this step is the application tuning model.

4.2 Runtime Application Tuning

This section first presents the overall behaviour of the tuning system during the runtime of an application in production mode, i.e., during the Runtime Application Tuning (RAT) stage of the READEX methodology. This is followed by descriptions of the main scenario based tuning phases in the RAT workflow (Section **??** through Section **??**). Further details on the implementation of the scenario detection and switching mechanisms can be found in Section **??**.

4.2.1 RAT overview

Figure **??** presents a high-level abstraction state diagram for the different tuning steps performed at runtime. An application run starts with the initialisation of the application and the underlying system software. The RRL is initialised upon the first event triggered by the application in the *Phase external computation* state During initialisation, the RRL reads the tuning model created during design time and sets up the measurement infrastructure necessary to perform runtime tuning. In this state all computation not part of the phase region is executed. The execution time and cost of this application initialisation phase will be negligible compared to the computation performed during the main progress loop and thus can be ignored for the tuning.

When the application enters the phase region, i.e., it enters the main progress loop, the relevant *Enter phase* event handler is executed. Within the *Phase* state, the main progress loop body is executed from where significant and insignificant regions can be called. For insignificant regions, no tuning will be performed. The region instance is simply executed using the existing system configuration. When the application enters a significant region, the relevant *Enter significant region* event handler is executed before the *Significant region* state is entered. The system configuration may thus be changed before the significant region instance is executed. Upon leaving the *Significant region*, the relevant *Exit significant region* event handler is called before returning to the *Phase* state.

The application may go through multiple iterations between the *Phase* state and the *Significant region* state. When the application reaches the end of the main progress loop body the current phase region instance is exited and the relevant *Exit phase tuning* actions are performed. The application then returns to the *Phase external computation* state and subsequently a new phase will be entered. The exception to this is when the last phase has been executed, in which case any relevant application post execution is performed before the application terminates.

Figure **??** provides a more detailed view of the control flow of the runtime tuning and its interaction with the application. As a working example, the simplistic, but still realistic, program code presented in Listing **??** in Section **??** is used throughout this section.

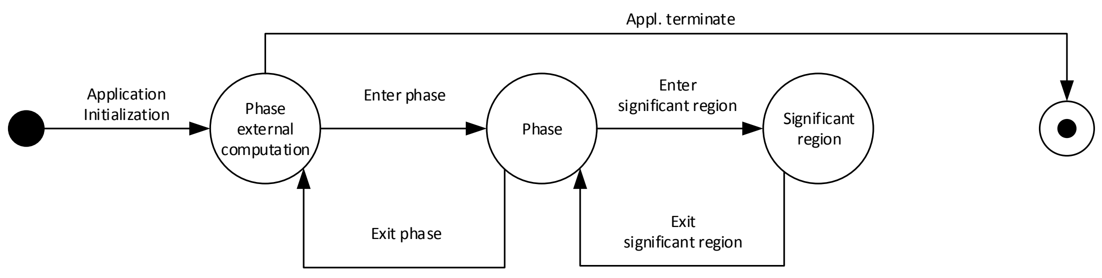

Figure 4.4: Runtime tuning state diagram depicting changes between states of the runtime library triggered by application events.

4.2.2 Initialization

A run starts when the application (MPI) processes are distributed across the nodes of the cluster. The application specific tuning model is used to initialise the application and the RRL. If available, the initialisation step also uses relevant application input data and node configuration information to determine values of input identifiers. The identifiers and their values together constitutes the input context elements, as defined in Section **??**.

The node configuration contains information regarding the underlying hardware architecture, including the number of CPU sockets, the CPU clock frequencies, number of accelerators, etc. Together with the application input data, this influences the degree of dynamism in different processes and the identifier values stored in the corresponding context elements will later be used during scenario detection.

During initialisation, objective measurement procedures are prepared that will be performed during later phase and significant region execution. The choice of objective function is made by the user during DTA stage and is typically expected to be either the overall energy consumption of the application or the energy delay product. Objectives such as total cost of ownership may also be used in the future. Finally, the initialisation step includes any application initialisation required prior to starting the main progress loop. In Listing **??**, this would be the initialisation of experiment variables.

4.2.3 Enter phase

Immediately after initialisation the application will start the execution of the main progress loop, i.e., the phase region of the application, which corresponds to the `for`-loop in Listing **??**. Each instance of this phase region starts by executing a *Enter Phase* pre-processor macro, which is added by the user to mark the beginning of a phase (see Section **??**). This macro triggers the event handler for this for the *Enter Phase* event. As described in Section **??**,

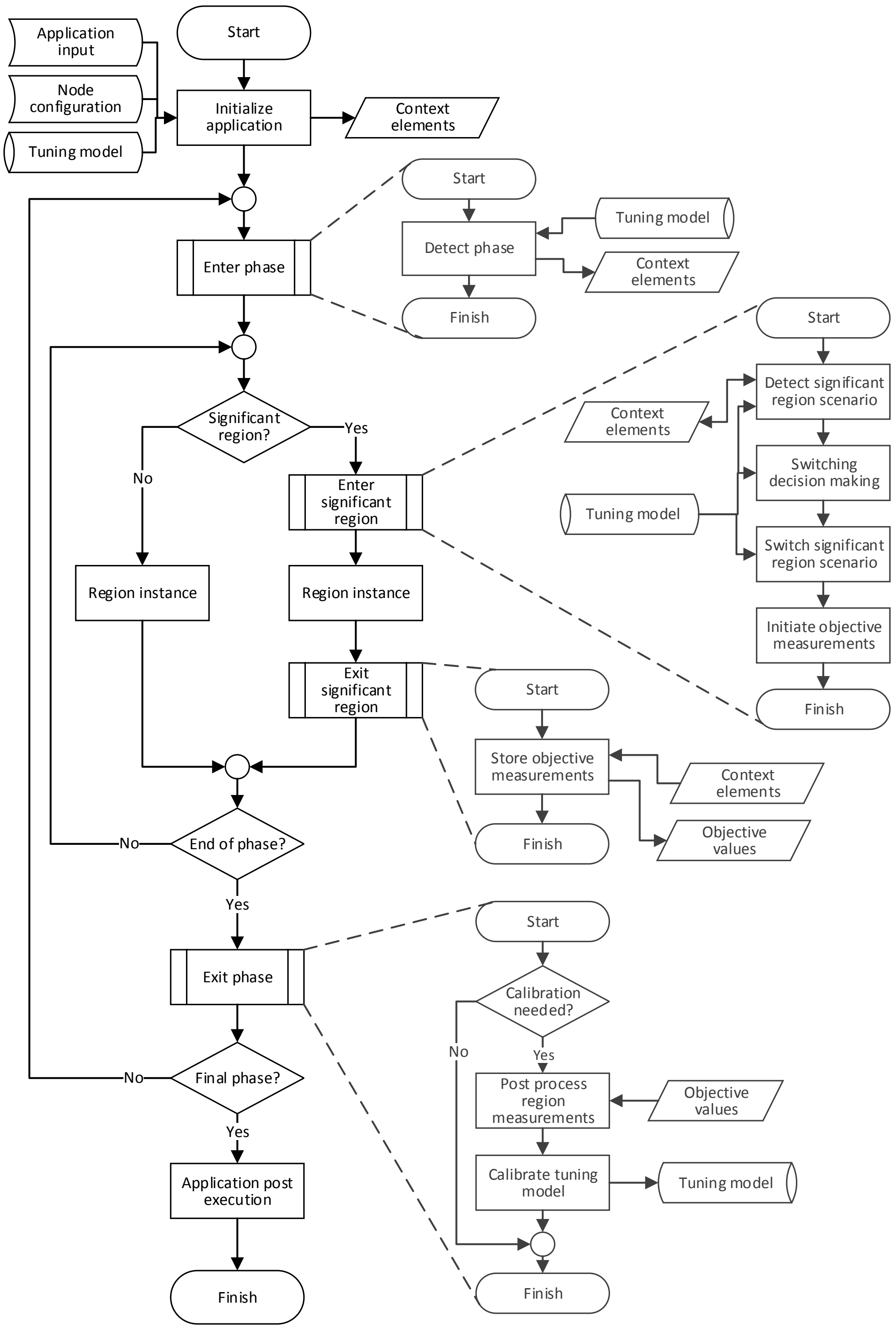

Figure 4.5: Runtime tuning workflow

phase identifiers may be used to distinguish different behaviour across phases. The event handler evaluates phase identifiers defined by the tuning model and stores the resulting context elements for use during subsequent significant region switching.

Within the phase region, there are a number of significant and insignificant regions. Plain code in the phase region itself, such as the `num_iterations++` statement in Listing **??**, will be regarded as insignificant. Functions called in the phase region or further down in the function hierarchy, can be insignificant or significant regions, as outlined in Section **??**.

4.2.4 Enter significant region

When a significant region is entered, the relevant region identifiers are evaluated by the event handler according to the tuning model and the corresponding significant region context elements are stored. Using the total set of context elements, i.e., input, phase, and significant region context elements, the upcoming significant region scenario is detected and a decision is made for the corresponding system configuration switching. The detection and decision are made according to the scenario definitions in the tuning model using classifiers and selectors, as described in Section **??** and Section **??**.

If a trade-off between conflicting objectives is required, e.g., between performance and energy consumption, an evaluation based on Pareto curves may be applied. It is, however, essential that the overhead of these operations remains minimal. In most cases, it is sufficient to use the context elements to perform a simple look-up in a table with precomputed values from DTA.

To be able to further fine-tune and calibrate the tuning model at runtime, a set of objective measurements are performed during execution. To be able to correlate the measurement values with the current identifier values, the current set of context elements are stored for later use when the current phase is exited, as described in Section **??**.

In some rare situations, e.g., if the design time analysis does not cover all possible context element combinations, a previously *unseen rts* can be detected. An rts is defined as unseen if the tuning model classifier is unable to link the current set of context elements with a specific scenario. This can be handled through a default platform and application configuration or through the selection of the scenario with context elements closest to the unseen rts.

4.2.5 Exit significant region

After the execution of a significant region, an exit handler is executed, which gathers the objective measurement results. These are linked with information about the scenario and configuration used during the execution as well as the context elements that caused the selection of the scenario. The complete data set is stored for later use as part of a scenario tuning model calibration. If the executed significant region instance was a previously unseen rts, the measurements are stored with an additional flag indicating that calibration should be performed at the next phase exit.

4.2.6 Exit phase

At the end of a phase region, an exit phase macro triggers the execution of an exit phase event handler. The main goal here is to perform a calibration of the tuning model. To keep the overhead as low as possible, this is only done when needed, for example, every ten phases or if a given objective criteria has been met, e.g., an unexpected increase in energy consumption for a given significant region.

When executed, the intent of the calibration step is to improve the significant region scenario detection and switching decision making in later phase regions. The calibration starts by performing a post processing of the results of the objective measurements gathered during all significant region executions since the last calibration. This may for instance reveal that different executions of a given significant region currently placed within the same scenario consumes significantly different amounts of energy. Such an observation indicates that the current set of scenarios may be suboptimal and a calibration of the tuning model can be beneficial. The calibration may include updates of the platform and application configuration for specific scenarios or changes to the relations between scenario identifier values and scenario selection. If the calibration is caused by a previously unseen rts, the rts is classified based on its context elements and the objective measurements performed during its execution. With this information, a scenario is selected to be used when the rts is encountered next. Depending on the objective measurement results obtained the next time the rts is encountered, it is either permanently placed in this scenario or a process is started of testing out alternative scenario placements over the course of the following execution cycles. A completely new scenario may also be generated, the circumstances and implications of which will be investigated in detail in later stages of the READEX project.

An additional calibration target will be the handling of load-imbalances, e.g., due to differing input characteristics. As much as possible will be handled at design time, for example, through the specification of domain knowledge as described in Section **??**. This could include the exposure of input data that determines the load of individual threads or processes. Using this information, the expected imbalances can be predicted and counter-measures such as processor speed adjustments can be used for re-balancing, e.g., as described in Section **??**. By incorporating this information into the tuning model during DTA, additional analyses at runtime are not required.

However, there might be unexpected imbalances that can be caused by hardware performance differences or unknown or unexpected input, e.g., if no domain knowledge specification is provided by the user. In this case, the calibration could consist of an evaluation of the wait states during the last phase execution and an update to the tuning model. This can make some processes execute faster than others to rebalance execution and minimise CPU cycles spent waiting for synchronisation. For the wait-state analysis, no global view on the application execution is required. However, as mentioned in Section **??**, there might be cases in which a global synchronisation is required to determine the parameter settings used throughout the next iteration.

4.3 Domain knowledge specification

The READEX tuning approach can be supported by domain knowledge specified by the code developer. An obvious example of domain knowledge is the phase region identifying the computation of the application phases. The user might also be able to provide additional identifiers for the significant regions, the phases, and the application input. This would improve the application tuning model significantly allowing to distinguish rts's with different characteristics.

The following list presents the domain knowledge entities that the READEX tool suite will understand and the way in which this information can be specified through Score-P annotations:

Phase Region: Automatic detection of the phase region is almost impossible without parsing and analysing the source code of the application. The phase region can be a single routine, a for-loop, a while-loop, or be constructed by jumps. On the other hand, developers of the code know where the phase region is and can easily provide this information. Therefore, READEX will allow the user to specify the phase region as shown below.

```
#include "SCOREP_User.inc"

SCOREP_USER_REGION_DEFINE(R1)
! phase region starts
for(step=1... max_iter)
  SCOREP_USER_OA_PHASE_BEGIN(R1, "OP",
      SCOREP_USER_REGION_TYPE_COMMON)
  time=time+dt
  computation(time)
  ...
  SCOREP_USER_OA_PHASE_END(R1)
! phase region ends
END for
```

SCOREP_USER_REGION_DEFINE(R1) defines a user region handle named R1. The phase region is then surrounded by SCOREP_USER_OA_PHASE_BEGIN and SCOREP_USER_OA_PHASE_END. The *online access phase region* enables external tools to configure Score-P dynamically when a phase is started. This configuration mechanism is used in DTA to perform experiments for evaluating different system configurations.

Both, the start and the end of the phase region entail a barrier synchronisation of all participating processes when an online tool like PTF is connected to Score-P.

Significant Regions: Other regions than the Score-P default regions, i.e., compiler-instrumented application functions and OpenMP regions, can be defined as user-defined regions with Score-P macros and can thus be provided as candidates for significant regions. These macros can enclose arbitrary code and are instrumented automatically. As a result, the Score-P monitoring library can handle these user-defined regions in the same way as any other region.

```
#include "SCOREP_User.inc"

SCOREP_USER_REGION_DEFINE(R1)

SCOREP_USER_REGION_BEGIN(R1, "OP", SCOREP_USER_REGION_TYPE_COMMON)
  foo(...);
  bar(...);
SCOREP_USER_REGION_END(R1)
```

Region Identifiers: As well as the region id and the call path, additional region identifiers can be used to distinguish runtime situations. These identifiers are specified as Score-P parameters for parameter-based profiling and can be of type integer and string. The following example demonstrates the use of Score-P parameters in a compiler-instrumented region.

```
void foo(int64_t myint, uint64_t myuint, char *mystring)
{
SCOREP_USER_PARAMETER_INT64("myint",myint)
SCOREP_USER_PARAMETER_UINT64("myuint",myuint)
SCOREP_USER_PARAMETER_STRING("mystring",mystring)
// do something
}
```

Phase Identifiers: Phase identifiers characterise the differences of phases and can be provided in the same way as a region identifier as Score-P parameter that are attached to the phase region.

Input identifiers: The READEX methodology will not only tune the application for a single input but will learn from running the application for different inputs. To be able to distinguish rts's with different characteristics in these runs, input identifiers are required. They should identify values from the input that characterise the variations, such as the grid size of the application domain. Input identifiers could be specified in the same way as region identifiers based on Score-P parameters.

Application tuning parameters: As outlined in Section **??**, the application can provide tuning parameters that switch the control flow. These tuning parameters are variables in the application that can take certain values. The address of these variables will be exposed to the RRL via additional Score-P macros.

5 READEX Tool Suite: Integrated Architecture

The READEX tool suite will support two major stages as described in Section **??**, i.e., the Design Time Analysis (DTA) and the Runtime Application Tuning (RAT) stages. To accomplish this, the tool suite will consist of two major components: an extension of the Periscope Tuning Framework (PTF, see Section **??**) used to control the DTA and a new READEX Runtime Library (RRL, see Section **??**). A third component, Score-P, will serve as a common infrastructure for PTF and the RRL, providing instrumentation and measurement capabilities to both components.

All tools will be mainly developed in C/C++ to allow for ease of collaborative development and distribution. A set of early prototypes will be developed and released to spark interest in the project and minimise risk, as described in Section **??**. A software quality assurance plan described in Section **??** has been introduced to ensure high quality software is developed throughout the project.

5.1 Design time analysis with PTF

The DTA stage will be executed by PTF, where the architecture of PTF and its integration with the RRL is shown in Figure **??**.

PTF Version 2.0 will be the starting point for the implementation of the DTA. It supports tuning applications at design time via tuning plugins [**?**]. The tuning plugins determine best system configurations for a given tuning aspect based on expert tuning knowledge, standard search algorithms, and experimental evaluation. Experimental evaluation is based on online configuration of the RRL through Score-P. The RRL will provide the appropriate mechanisms for the execution of tuning actions for the rts's.

The main existing components of PTF are shown in black and orange in Figure **??**. The orange components will be extended in READEX. These components are:

The Analysis Component provides analysis strategies for tuning plugins. These strategies can be used to collect performance measurements in the form of performance properties. New DTA analysis strategies will be provided as required.

Plugin Control triggers the individual steps in the READEX tuning plugins and provides the required data structures to the plugins.

The Performance Database stores the performance data gathered during the execution of DTA experiments. Plugins access this information via performance properties.

Search Algorithms provided in PTF and are used to automatically evaluate system configurations from a search space constructed by the tuning plugin.

The Experiments Engine automatically executes experiments to evaluate certain system configurations. It configures the RRL to assign values for the tuning parameters given

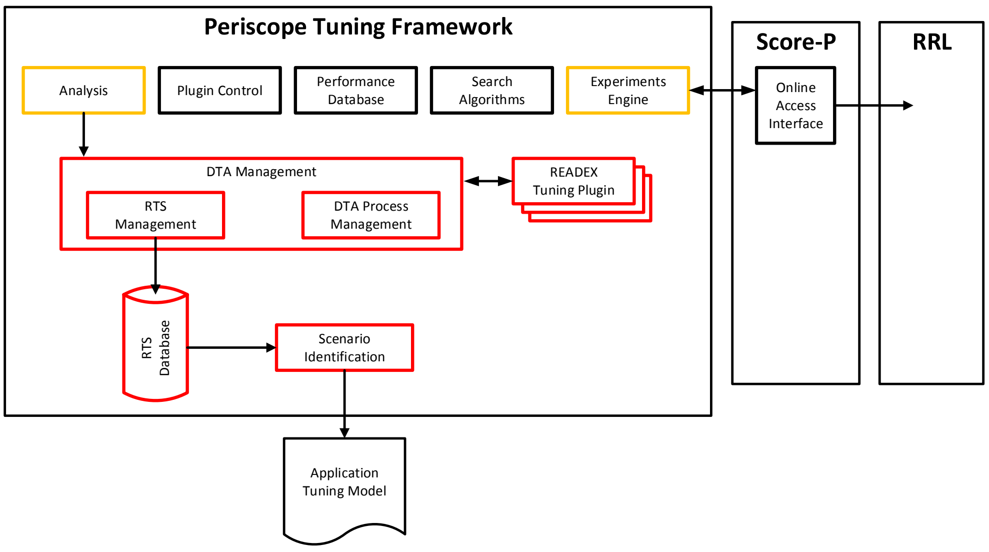

Figure 5.1: Extensions to PTF implementing READEX DTA.

by the system configuration. The experiment engine will be extended to support rts's with their significantly extended context in READEX.

In addition to these base components, PTF will be extended by the addition of READEX specific components. These are shown in red in Figure **??**. The following components will be added to PTF:

DTA Management: will control the overall execution of DTA. For this purpose, it includes two subcomponents. **DTA Process Management** will interact with the other PTF components to execute the workflow presented in Figure **??**. It will collect required information via analysis strategies and trigger both tuning cycles and the READEX tuning plugins. It is responsible for incrementally collecting information about rts's into the RTS Database and for triggering the final creation of the tuning model. The second subcomponent is the **RTS Management** that will manage the RTS Database.

RTS Database: collects information about the rts's of significant regions including their selector.

READEX Tuning Plugins: For each tuning aspect presented in Section **??** a tuning plugin will be created that determines the best configuration for an rts and the selector. These tuning plugins will be triggered during intra-phase analysis.

Scenario Identification: constructs the final application tuning mode. It analyses the rts's and aggregates them into scenarios with a corresponding classifier according to their context and selector. It generates the application tuning model into a file in XML format.

The arrows shown in Figure ?? present the data flow between the components. The analysis services provide the tuning potential, the DTA hints, and the significant regions. The READEX tuning plugins receive the list of significant regions as well as rts results obtained in previous tuning cycles. They provide the rts's that have been found, as well as their selectors, to the DTA Management. The rts information in the RTS Database is inserted by the DTA Management and used by Scenario Identification for the generation of the tuning model. Finally, the experiments engine configures Score-P with measurement and tuning requests for rts's to evaluate different system configurations.

5.2 Runtime application tuning with the RRL

Figure ?? depicts the RRL infrastructure, showing existing and to-be-developed components in Score-P and the RRL itself, which will be implemented by means of a substrate plugin by way of a **Substrate Plugin Interface**. This plugin interface is currently being developed as part of the project and allows for generic access to measurement data at runtime without direct integration into Score-P. This approach acts to reduce maintenance and integration efforts by keeping the RRL as a separate entity. As a substrate plugin, the RRL will thus have access to all measurement data gathered through various means of instrumentation, where it can use this information to make switching decisions based on the tuning model created at design time.

Within the RRL, the following components for scenario detection, switching, and calibration will be implemented:

Scenario detection will determine the upcoming scenario based on the current rts, e.g., by monitoring of significant regions, their call hierarchy, and input parameters, as well as information on the current phase of the application. The mechanism to be implemented in the RRL will be generic in order to apply it to any target application. The scenario detection component will employ the classifiers created during DTA and stored in the tuning model to map an identified rts to a scenario. The scenario detection will incorporate information on the application input as well as on the current phase region into the scenario classification.

A general scenario switching mechanism that can be used across applications will be developed and included in the RRL. Switching of parameter settings will be performed via the **parameter control plugins** that are also employed by the DTA components. Efficient switching decision making requires a low-overhead mechanism, e.g., a lookup table that binds parameter settings to a scenario identifier and is generated from the tuning model developed during DTA. The scenario switching mechanism will also be employed by PTF for DTA to test different parameter settings for a given rts.

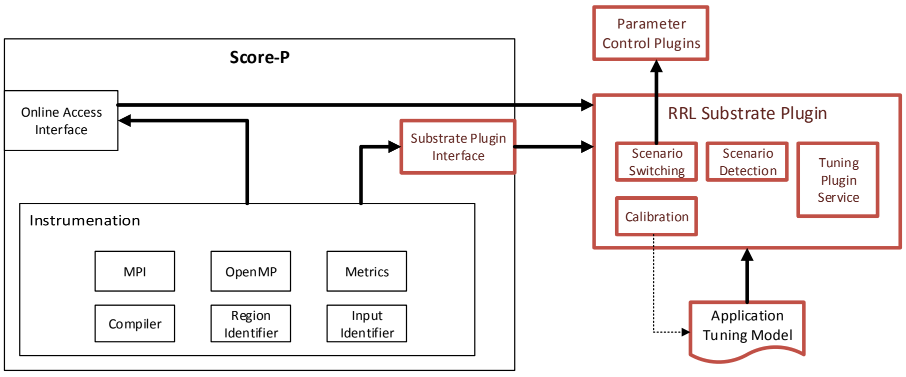

Figure 5.2: Score-P infrastructure (existing components are coloured black; red components will be subject to research and development during the READEX project).

Calibration of the tuning model will be performed by a calibration component, which will be able to adjust the tuning decisions at runtime and eventually feed back the results into the tuning model at the end of the application run. If required, the calibration process can perform global inter-process communication operations, e.g., to synchronise global parameters or to determine global objective values.

In addition to the components of the RRL described above, the READEX project will also develop new **parameter control plugins**, which provide a convenient way of switching parameters of different tuning aspects, as described in Section **??**. They will be employed by both the READEX tuning plugins during DTA and the scenario switching component during RAT.

5.3 READEX tool suite prototypes

This section describes the different versions of the READEX tool suite prototypes in terms of their scope and incremental feature implementation. The goal of the project is to release versions of the READEX tool suite every six months, starting from M12. This approach will help to identify potential challenges as early as possible so that they can be addressed in a timely manner. The key deliverables for the tool suite include the following:

- D4.2: Prototype READEX tool suite – M18
- D4.3: READEX tool suite version 2 – M30
- D4.4: Final READEX tool suite – M36

Below, we summarise the different tool-suite prototype versions in terms of their feature support for software components relevant to the DTA and RAT stages, respectively.

5.3.1 Pre-alpha version (M12)

A pre-alpha version of the tool-suite prototypes will be released as an implementation of the basic workflow of the tool-suite. A pre-alpha version in M12 will be kept internal to the project and will ensure the early availability of a working prototype of the tool-suite.

DTA As a first step for the DTA implementation, a mechanism for significant region analysis will be implemented along with the support for intra-phase dynamism handling. Tuning for a single objective, e.g., energy efficiency, will be supported via a READEX tuning plugin. Region IDs will be used as a classifier to aggregate multiple rts's into a scenario, i.e., scenario identification. As a final step towards DTA implementation, a phase tuning model including the classifier and selector will be implemented, where the selector will specify a single best configuration for each scenario.

RRL For the first prototype of the RRL, a parameter control plugin will be implemented to support the tuning actions for the CPU frequency. Based on the tuning model produced by the DTA stage, a scenario detection mechanism based on the region ID will be implemented. Once a scenario is detected, the best configuration for the scenario will be selected using the selector within the tuning model and specified tuning actions will be executed, i.e., scenario switching. The scenario switching mechanism will be implemented as part of the pre-alpha release of the tool suite.

5.3.2 Alpha version (M18)

An alpha version of the READEX tool suite will align with D4.2 and will be available in M18. The alpha version will build on the previous version of the tool suite and will extend the pre-alpha version in terms of the following features:

DTA In terms of the DTA extensions for the alpha version, additional identifiers such as region call paths and region parameters will be supported as region identifiers. Additional READEX tuning plugins will be made available to support tuning for multiple aspects. Scenario identification and the phase tuning model from the pre-alpha version will be extended with additional classifiers based on the availability of the additional identifiers, i.e., call path and region parameters.

RRL The alpha version of the tool suite will include additional parameter control plugins to support multi-aspect tuning actions. Scenario detection mechanisms will be extended for new identifiers within the tuning model produced by the DTA stage. In terms of scenario switching, more complex selectors and switching mechanisms will be implemented, e.g., to handle multiple tuning aspects.

5.3.3 Pre-beta version (M24)

A pre-beta version of the prototype will be released internally to the READEX consortium at M24.

DTA In terms of the DTA stage, inter-phase dynamism will be supported along with the phase identifiers, in addition to the previously supported intra-phase dynamism and region identifiers. In terms of support for multiple tuning aspects, the final set of READEX tuning plugins will be available as part of this prototype. A preliminary version of the domain knowledge specification to leverage user domain knowledge will also be integrated. A phase-aware tuning model incorporating phase identifiers for scenario identification and more complex selectors will be supported.

RRL In terms of the RRL extensions, a final set of tuning plugins will be made available to execute multi-aspect tuning actions. Scenario detection at runtime will be extended to handle phase scenarios, identified within the phase-aware tuning model. For scenario switching, a global decision making mechanism will be implemented to support more complex selectors and global synchronisation for tuning actions.

5.3.4 Beta version (M30)

A beta version of the tool-suite will align with D4.3 and will be available in M30.

DTA The DTA component will be extended to support multiple application inputs as an additional form of application dynamism. Moreover, visualisation capabilities will be provided to analyse application dynamism along with the tuning potential and hint analysis features. Scenario identification will be extended to incorporate application input based identifiers. A final version of the READEX programming paradigm, i.e., domain knowledge specification, will be integrated. Finally, support for the application tuning model will be available as part of this version of the tool suite.

RRL For the RRL, the scenario detection mechanism will be extended to support an application input based classifier for identification of scenarios. In terms of new features, a preliminary implementation of the calibration mechanism will be available to optimise the tuning model at runtime and to support unseen rts's.

5.3.5 Release candidate (M36)

A final version of the READEX tool suite will align with D4.4 and will be available in M36. Final versions of the DTA and RRL will be made available as the Release Candidate (RC) for the READEX tool suite.

5.4 Software Quality Assurance Plan

This section outlines the common best practices for the development of software products in the scope of the READEX project. The objective of the software quality assurance plan is to ensure the following:

- Robustness of software through test-driven development and component reviews.
- Maintainability using automated builds and release management.
- Extensibility with well-defined configuration management.

In the subsequent part of this section, individual components of the software quality assurance plan will be discussed.

5.4.1 Component Reviews

Component reviews in the context of the READEX project involve assignment of reviewers (ideally software developers) to the software deliverables. Component reviews have two primary goals:

- Ensure that the individual functional requirements for a given software component are met in compliance with the software specification.
- Verify that the required interfaces for a software component are implemented correctly to avoid any integration problems.

These goals are to ensure that developed software is functional and can be integrated with other components. Lightweight black-box testing will be used for component reviews.

5.4.2 Test-Driven Development

Test-driven development has become the de facto standard for quality development of software products. Software testing cannot guarantee functional correctness but can increase confidence that systems will perform without failure. Testing can facilitate the debugging process by locating faults that then can be fixed early in the development process. Test-driven developments serves two important purposes:

- Detect system errors at an early development stage when they are least expensive to fix.
- Evaluate whether a program is usable or not. Since testing can only determine the presence of errors and not the absence of them, developers have to balance the trade-off between the level of testing and the number of test cases to ensure cost effectiveness. Well thought out test cases will increase the robustness of the READEX software.

Testing should be considered an integral part of the software development process throughout its life cycle. In general, software testing can be employed at various different levels. The relevant levels for the READEX software development process include the following:

Unit Testing is the lowest level testing of individual units of code, e.g., a function, class, or package. Individual unit tests are carried out in isolation from other units and verify that the defined design has been correctly implemented. The READEX component developer will write and perform unit tests. In the scope of the READEX project, individual component developers will be responsible for unit testing of their software components. These unit tests can then be included in the build scripts for the automated testing during the component reviews.

Integration Testing is the testing of the interfaces between individual components. READEX component reviewers will be responsible for integration testing. WP4 will be responsible for the integration testing of the READEX tool-suite. If certain units are not yet available for testing, stubs will be used to simulate their functionality. Inputs and outputs may be hard-coded or read from a file. Within the READEX project continuous integration will be employed using the Jenkins[4] Continuous Integration (CI) system to oversee the integration and flow of development, testing, deployment, and support. It is envisaged that the Jenkins CI will be employed from the early stage of the READEX project to facilitate automated build and release management.

Use-case Testing describes testing according to the expected usage of the system. The application and validation work-package, i.e., WP5, will be responsible for the use-case testing of the READEX tool suite.

5.4.3 READEX Repositories

Repositories are vital for data management during the lifetime of the READEX project. The following repositories have been set up for the lifetime of the project:

Source Code Repository: A Git[5] repository is used for source code management. Git allows for sophisticated branching to streamline the development process among the distributed development teams. The repository will be centralised to ensure project-wide accessibility. All the relevant software documentation as well as technical and scientific documents are to be stored in the source code repository.

Application Repository: A separate Git repository will be used to manage applications of interest. These applications will be used to test the READEX tool-suite.

Data Repository: Since application data may require significant storage space, it will be managed in a separate data repository, i.e., the project storage on the Taurus system installed at TU Dresden, which is accessible for all project members.

[4]`https://jenkins-ci.org/`
[5]`https://git-scm.com/`

Document Repository: As described in deliverable D7.1, a Sharepoint project has been set up that contains all presentations as well as management and and other non-technical documents. The repository is available at `https://sharepoint.tu-dresden.de/projects/readex/` and only accessible for project members.

5.4.4 Build and Release Management

Build and release management is the process of managing software builds, e.g., nightly, weekly, or fortnightly, and releases from development stage to software release. Due to the distributed nature of the development model in the READEX project, a unified (possibly automated) build and release management will be required to seamlessly build, test and integrate various different components before making an official release. For this purpose, the Jenkins continuous integration system will be used, as described in Section **??**.

Another important element of the release package is the documentation for all relevant users of the system, i.e., developers, administrators and end-users. Hence, software release will include a copyright statement and licensing information as well as documentation on how to deploy and use the software.

All software releases will be published through the READEX website at `http://www.readex.eu/index.php/dissemination/software/`.

5.4.5 Anomaly Management

The detection and handling of software anomalies are an integral part of a software development process. The READEX project has defined procedures to manage and resolve anomalies identified through the test-driven development process described in Section **??** and beyond official releases. The procedure will be relevant from the integration phase onward.

An anomaly is generally defined as an error or bug in any item that does not conform to the specified requirements or specification. An anomaly or bug can be major and thus render the system unusable or minor and affect functional use in only a few cases. The person discovering an anomaly is responsible to report the bug and may choose to directly provide a fix if possible. The person is responsible for reporting the bug to the respective work package (WP) leader using the project-internal ticket system, which allows other WP leaders and developers to be included in the communication if appropriate. The WP leader is responsible for fixing or reassigning the bug to a developer or other WP leader he thinks is more appropriate. A timely resolution of anomalies is desirable to mitigate the impact of the anomaly on the development process and/or the users.

6 Application Case Study

This section presents an application case study, which provides a supportive statement for the READEX approach, as outlined in previous sections. Using a model cube benchmark, we discuss the energy consumption evaluation of a Finite Element Tearing and Interconnecting (FETI) solver. These solvers are used to solve extremely large systems of linear equations on HPC clusters. The measured characteristics illustrate the behaviour of various pre-processing and solve phases related mainly to the CPU frequency.

6.1 FETI methods and their implementations

Partial Differential Equations (PDEs) are often used to describe phenomena such as sheet metal forming, fluid flow and climate modelling, where the computational approaches taken to finding the solution to such PDEs involve solving a large system of linear equations.

When scientific applications solve PDEs that are too big to fit in the memory of a single machine or demand more processing power than a single machine can deliver, Domain Decomposition Methods (DDMs) are required. DDMs are used to divide the original problem into smaller sub-domains that are distributed across the compute nodes of a HPC cluster. Without going into too many details, it suffices to say that the decomposition factor H defines both, sub-domain size as well as the total number of sub-domains for a problem of given size. The size of a problem is determined by the discretization parameter h, which defines the mesh granularity. Parameter H acts as an important application parameter.

The Finite Element Tearing and Interconnecting (FETI) method forms a subclass of DDM, as it efficiently blends Conjugate Gradient (CG) iterative solvers and direct solvers. Since the number of CG iterations is independent of the discretisation parameter h, the FETI method has both the parallel and numerical scalability to scale to tens of thousands of processors. IT4Innovations is developing two in-house FETI-based software packages. The first one is called PERMON [**?**], the second one is ESPRESO [**?**]. Both PERMON and ESPRESO focus on engineering applications.

The two main phases of the FETI method are *pre-processing* and *solve*. In the pre-processing stage, the stiffness matrix K is factorised and the natural coarse space matrix G and coarse problem matrix GG^T are assembled. The latter matrix is also factorised. Both of these operations are considered to be among the most time consuming and thus among the most energy consuming operations.

The solver employs the Conjugate Gradient (CG) algorithm, which consists of Sparse Matrix-Vector Multiplications (SpMV), vector dot products, or AXPY functions. For each iteration, it is necessary to apply the direct solver twice, i.e., forward and backward solves for the so-called pseudo inverse action and the coarse problem solution. All these operations are covered by the basic sparse and dense BLAS Level 1-3 routines, which allows us to explore their different computational characteristics.

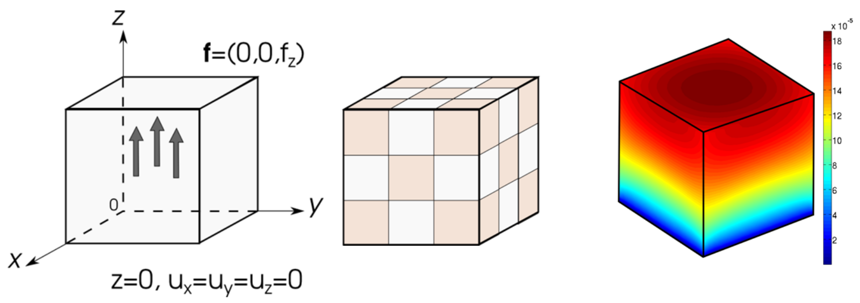

Figure 6.1: Model 3D linear elastic cube.

6.2 Model problem

As a benchmark, the 3D linear elastic cube was used with the bottom face fixed and the top one loaded with a surface force, as depicted in Figure **??**. For these computations, a mesh is generated and decomposed into sub-domains by the PermonCube benchmark generator. The parallel mesh generation is controlled by two groups of parameters. The number of subdomains $N_S = XYZ$ is managed by parameters X, Y, Z and similarly the number of elements per subdomain is given by x, y, z (both considered in the respective axis directions). The numerical scalability of the FETI method within PERMON is characterised by the number of iterations decreasing with increasing N_S (decreasing H) for the fixed h parameter. Decomposition into a large number of sub-domains favourably affects the time of factorisation of K and the pseudo-inverse K^+ action. However, it also increases the size of the coarse problem, whose solution becomes a critical part of this method for a large number of sub-domains.

6.3 Experimental setup

As described in Section **??**, an initial set of tuning parameters has been identified for the READEX project. Although several parameters have been investigated so far, we will concentrate on the CPU frequency for this early-stage case study.

As part of this study, we placed HDEEM measurement functions around individual application regions, which allow for energy measurements on the Taurus system at TUD. By instrumenting the code with HDEEM at appropriate locations, we are able to measure the energy consumption of either the whole solver or selected regions only. For example, for the case of the finer grain tuning and measurement, we can place the switching points before and after the functions of interest to change the frequency and measure the objective value. Thus, the function names identifying the regions implicitly serve as identifiers for the tuning of the CPU frequency. The instances of the function executions determine various rts's. The sequence of these rts's forms the application execution, e.g., the pre-processing step followed by the solve step.

The measured power consumption of the particular rts's under the given system configuration are depicted as objective functions and are illustrated in Figures ??, ?? and ??. As part of this early-stage investigation we aim to indicate a function mapping of rts's to an optimal system configuration. Tuning relevant dynamism If the switching of between system configurations shows an improvement in the objective function, e.g., a reduction in the energy consumption when running different regions with different CPU frequencies, in our FETI solver.

6.4 Preprocessing, solve and overall FETI vs. CPU frequency

The measurements of PERMON's FETI solver energy consumption on a single computational node were performed using the Linux CPUFreq utility[6] for frequency changing (cpufreq_set_frequency()) and HDEEM library version 2.1.5 for the accurate measurements.

The relation of the CPU frequency and the consumed energy for the complete solution of the problem is shown in Figure ?? for (i) the whole FETI (preprocessing and solve), (ii) the preprocessing stage, and (iii) the CG solver stage. Figure ?? also shows standard deviations based on repeated measurements where the mean values were computed from 10 repetitions.

Taking into consideration the energy savings using static tuning in Figure ??, we can see that the total saved energy for the whole FETI computation (i.e, the difference between energy used with default frequency 2.5 GHz and minimum energy used with frequency 1.7 GHz) is $\approx 10\%$. This is true if the solve runtime is similar to preprocessing time, which is the case of a small problem used in this study.

If we need to solve the problem more precisely or we need to solve time dependent problems, the preprocessing stage becomes negligible as it is performed only once at the start of the simulation and hence, the solve stage will start to dominate. In the solve stage, we can see from Figure ?? that the improvement in energy efficiency between using the default frequency 2.5 GHz and the best-found CPU frequency at 1.6 GHz, is $\sim 13\%$.

Comparing the best static tuning frequency 1.7 GHz to the dynamic frequency tuning with (i) 1.6 GHz for the solve phase and (ii) 2.1 GHz for the preprocessing phase, we can see that the dynamic switching approach provides only a 0.6% and 0.4% reduction in energy consumption, respectively. To achieve an improvement to energy efficiency, we we further investigate dynamic tuning at the finer grain of CG kernels.

6.5 CG kernels of the FETI solver

In order to further evaluate the dynamism in the FETI solvers we have performed energy consumption measurements of the main kernels of the solve phase, as shown in Figure ??. These kernels are: (i) Preconditioner action (lumped $M_L v$ or Dirichlet $M_D v$), (ii) Operator action Fv, (iii) Projector action Pv, and (iv) vector operations (AXPY). For the measured operations, we have employed the following libraries: PETSc for $M_L v$, the MKL for $M_D v$, PETSc + MUMPS Cholesky for Fv, PETSc + SuperLU_DIST for Pv, and the MKL for

[6]CPUfreq: `https://www.kernel.org/doc/Documentation/cpu-freq/index.txt`

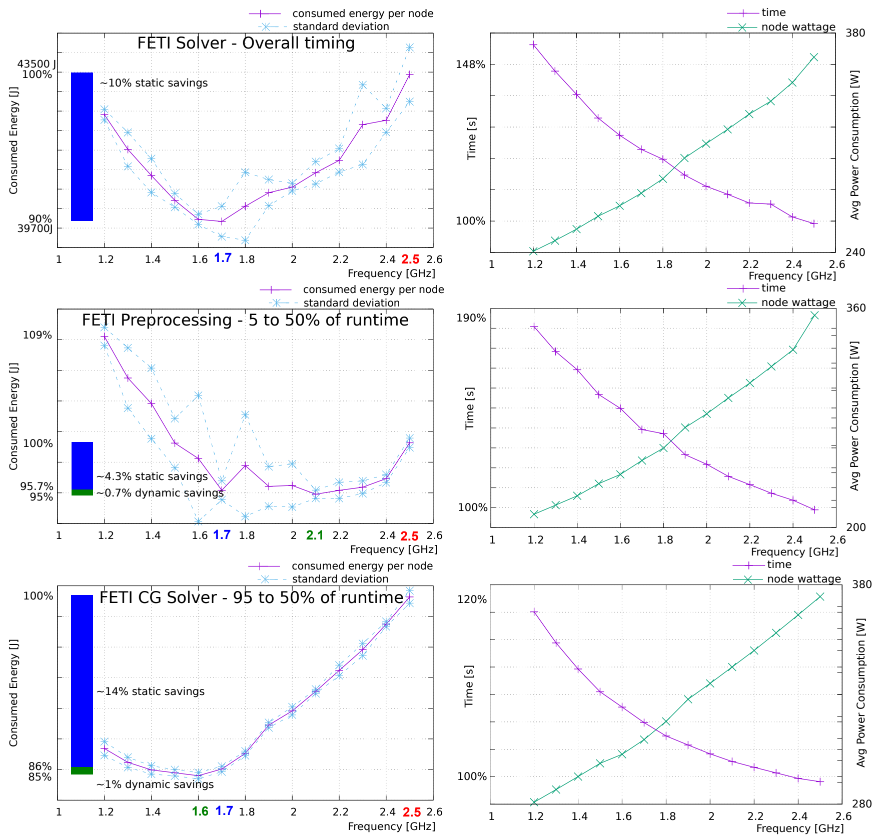

Figure 6.2: Dynamism evaluation for FETI solver for model problem with X=4; Y=3; Z=2; $x = y = z$=52; N_S=24; number of elements N_E=140,608; primal dimension N_P=10,719,144; kernel dimension N_K=144; 62 CG iterations (Red - default setting, Blue - best static configuration and associated energy savings, Green - best dynamic configuration and associated energy savings).

Table 7: Static and dynamic tuning savings of the FETI solve stage and both pre-conditioners. Overall savings are computed from the absolute savings and the runtime percentage. Dynamic savings are relative to the static tuning results.

Action	Runtime	Static Tuning		Optimal Frequency	Dynamic Tuning	
		Saving	**Overall**		Saving	**Overall**
$M_L v$	10%	3.2%	**0.32%**	2.1 GHz	3.1%	**0.31%**
Fv with M_L	80%	12%	**9.6%**	1.6 GHz	1%	**0.8%**
$M_D v$	40%	13%	**5.2%**	1.2 GHz	4%	**1.6%**
Fv with M_D	50%	12%	**6%**	1.6 GHz	1%	**0.5%**
Pv	6%	2%	**0.12%**	2.3 GHz	1%	**0.06%**
AXPY	4%	13%	**0.52%**	1.2 GHz	6%	**0.52%**
Total overall savings:						
with M_L			**10.56%**			**1.69%**
with M_D			**11.84%**			**2.68%**

vector operations. The optimal frequencies and the improvement to energy efficiency achieved for the individual kernels compared to running with the best-found static configuration, i.e, 1.7 GHz, are shown in Table **??**.

In order to evaluate the overall potential of the dynamic tuning inside a single iteration of the FETI CG solver, we have to take into account the execution times of the respective kernels. For example, although the vector operations can save up to 13%, this saving is small since it takes less than 4% of a single iteration runtime. The most time-consuming regions are the matrix-vector multiplications carried out by both the Fv operator and the Dirichlet pre-conditioner $M_D v$ as they each account for 50% and 40% of the overall runtime, respectively. The projector runtime is highly problem-dependant and is mostly influenced by the problem's number of sub-domains. For the problem used in our tests, its runtime is small. In order to measure its effect, we would have to run it on tens or hundreds of nodes. However, from our experience on large runs, the projector can take between 1-10% of the runtime.

The energy consumption characteristics of the projector operator will grow with the size of the coarse problem matrix. However, for this particular test case, performing dynamic tuning for this region does not yield any significant savings and is, therefore, an example of an insignificant region. As an example of application input parameter, the total number of sub-domains defines the projector size and its behaviour.

In Table **??** we summarise the overall savings achieved by the static and dynamic tuning. This table shows that based on the best-found static configuration we can save additional 2.68% by dynamic tuning of the CPU frequency as a single system parameter.

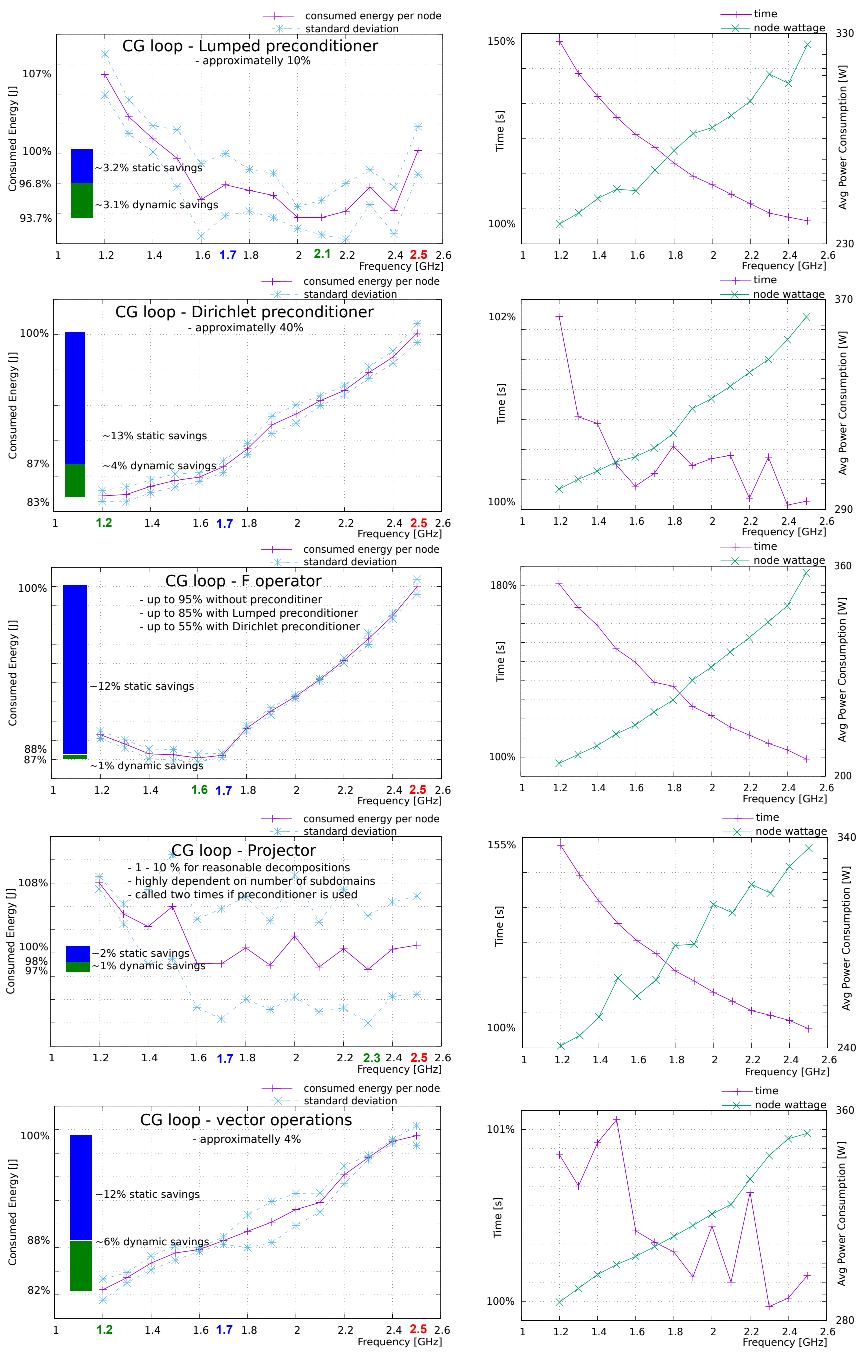

Figure 6.3: Dynamism evaluation of the solve phase, the CG solver in FETI: Fv, Pv, M_Lv, M_Dv and BLAS1 (vector) operations for model problem with X=4; Y=3; Z=2; $x = y = z$=52; N_S=24; number of elements N_E=140,608; primal dimension N_P=10,719,144; kernel dimension N_K=144; 62 CG iterations (Red - default setting, Blue - best static configuration and associated energy savings, Green - best dynamic configuration and associated energy savings).

6.6 Local Dense and Sparse BLAS evaluation

We have also analysed the performance of the local Matrix-Vector and Matrix-Matrix multiplications with dense and sparse matrices, i.e. BLAS 2 and 3 and Sparse BLAS 2 and 3 routines. As input data, we have assembled a FEM block diagonal symmetric positive semidefinite stiffness matrices K. Here, we have employed sparse MM and sparse MV from PETSc, dense MM and dense MV from Intel MKL.

Figure **??** illustrates the completely different behaviour of BLAS 2 and BLAS 3 routines with regards to power consumption. In case of the memory-bound dense Matrix-Vector multiplication, the minimum in energy consumption is reached for the lowest frequency of 1.2 GHz. In contrast to that, for the dense Matrix-Matrix multiplication the minimum in energy consumption is reached for the highest frequency of 2.5 GHz as it is more computationally intensive. A similar situation has been observed for sparse BLAS 2 and BLAS 3 routines, where the minimum of energy for sparse Matrix-Vector multiplication is reached for the lower frequency of 1.5 GHz while the minimum of energy for sparse Matrix-Matrix multiplication is reached for a higher frequency of 2.3 GHz.

In other words, for applications intensively and equally using (dense) BLAS 2 and 3 operations, we can improve energy consumption by $\sim 20\%$ by dynamically switching between optimised configurations. The savings for sparse BLAS 2 and 3 are not negligible and of course depend on the structure of the sparse matrix. While not depicted in this report, the behaviour of BLAS 1 routines is similar to BLAS 2 routines.

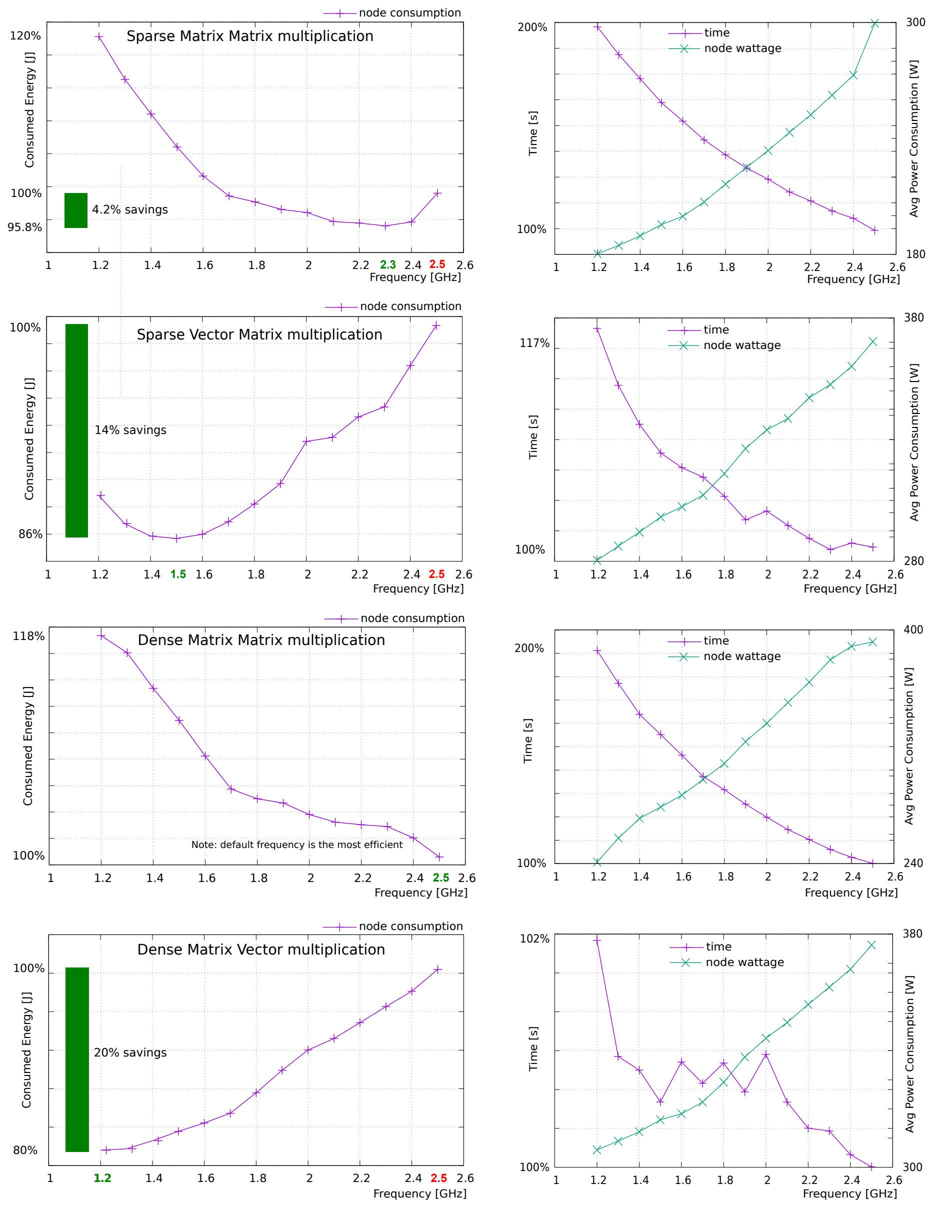

Figure 6.4: Relation of consumed energy and frequency (Intel MPI, gcc, O3) for 100x sparse Matrix-Vector multiplication Kv (Sparse BLAS 2) for blocks with 24^3 elements, 10x dense Matrix-Vector multiplication Kv (BLAS 2) for blocks with 16^3 elements, 1x sparse Matrix-Matrix multiplication KK (Sparse BLAS 3) for blocks with 24^3 elements, 1x dense Matrix-Matrix multiplication KK (BLAS 3) for blocks with 16^3 elements (Red - default setting, Green - best dynamic configuration and associated energy savings)

7 Summary

In this deliverable, we have presented the fundamental concepts of the READEX approach and have specifically presented the two-stage READEX methodology of Design Time Analysis (DTA) and Runtime Application Tuning (RAT). Along with this, we have described the concept of a tuning model and how this model is generated as the final outcome of the DTA stage of the READEX methodology. We have described in an abstract manner how the tuning model is subsequently employed by the RAT stage to guide dynamic runtime tuning of the HPC stack through the READEX Runtime Library (RRL), which will be developed as part of the project. We also describe how the fundamental concepts of the READEX methodology can be applied to define a tuning potential metric, which is central to the overall READEX methodology. Moreover, we have shown how the fundamental READEX concepts can be expanded to include the central idea of inter- and intra-phase dynamism as well as the concept of handling a multiplicity of application inputs. By presenting a formal mathematical description of the READEX concepts for the first time, one of the central aims is to minimise ambiguity in their definitions, to improve common understanding and communication between the project partners as well as to support the development of READEX DTA and RAT methodologies and software components throughout the lifetime of the project.

In our description of READEX-relevant tuning parameters, we have distinguished between three different levels of the HPC stack, i.e., hardware parameters, system software parameters, and application-level parameters. The tuning parameters that we have so far identified as being relevant to READEX include processor core and uncore frequencies, MPI and OpenMP runtime parameters as well as application-specific parameters. It has been continuously emphasised throughout that all of these tuning parameters can be influenced at runtime, a characteristic that is fundamental to the READEX approach. While all of the tuning parameters we describe in this deliverable are considered relevant to READEX, further investigations are required to confirm that the impact they have on the energy efficiency of applications in production mode is significant enough to warrant inclusion in the READEX tool suite.

We have followed our description of READEX concepts and tuning parameters with a description of the DTA and RAT approaches, which includes a detailed description of the workflows of each stage. In terms of the DTA stage we have specifically described the key steps of instrumentation, significant region detection, tuning model prediction, intra- and inter-phase analysis, hints analysis and tuning model computation, each of which are fundamental to the DTA workflow and will feature as software implementations in the READEX tool suite. Likewise, for the RAT stage, we have specifically described the key steps of initialisation, phase entry and exit and significant region entry and exit. When describing the overall READEX approach, we have described how the tuning approach can be supported by domain knowledge, specified by a target application developer or user. It is anticipated that the user/developer will be able to provide additional identifiers for significant regions, phases, and application input that will aid the READEX tool suite during analysis.

We have detailed how the tool suite will leverage two major software components: an extension of the existing Periscope Tuning Framework (PTF) used to control the DTA stage and a new READEX Runtime Library (RRL), which will be linked to the target application during the RAT stage. We have also described how Score-P will serve as a common infrastructure for PTF and the RRL, providing instrumentation and measurement capabilities to both components. In addition to a high-level description of how this software infrastructure will serve the READEX tool suite, we have provided an architectural description of the software components, including the extensions that will be implemented during the READEX project. On top of this, we have provided a description and release plan for READEX tool suite prototypes as well as a plan to ensure software quality throughout the development phases of the project.

Finally, we have described an application case study that centres on the PERMON and ESPRESSO applications, which focus on Finite Element Tearing and Interconnecting methods, used widely in the engineering domain. As part of this case study, we have already carried out energy consumption measurements as a function of tuned processor frequency on the Taurus system using instrumented versions of each of these applications. On top of these early stage results, we have also identified potential dynamism in each of the applications, which can be exploited in the READEX methodology as well as application-specific tuning parameters that we hope to target in the near term of the project.

In developing the READEX methodology and tool suite, it is also worth finally mentioning its limitations from the perspective of both methodology and design. Firstly, the tool suite will target only C/C++ and Fortran applications. Since the majority of HPC applications are written in these languages and since READEX is mainly targeting the HPC community, this is not regarded as a severe limitation. By the nature of the fundamental theoretical concepts outlined in this deliverable, it is clear that the tool suite can only aid in the improvement of performance and energy efficiency if the the target application exhibits some form of phase-like dynamic behaviour, i.e., what we have referred to as phase dynamism in this deliverable. Likewise, the tool suite will need to have access to various identifiers that can help to properly expose and characterise this dynamism. Finally, there is the challenge of carrying out the DTA and RAT stages at different scales on a given HPC system and, if this is viable at all, to find out what amount of flexibility is required for doing so. To determine this more clearly, and to surmount any challenges, will require further investigations during the early stages of the project.