GA no. 671657

D4.5

Final Description of the READEX Programming Paradigm

Document type:	Report
Dissemination level:	Public
Work package:	WP4
Editor:	Madhura Kumaraswamy (TUM)
Contributing partners:	TUM, Intel
Reviewer:	Kai Diethelm (GNS)
	Venkatesh Kannan (ICHEC)
Version:	1.3

Document history

Version	Date	Author/Editor	Description
0.1	22/06/17	Madhura Kumaraswamy (TUM)	1st draft
0.2	19/08/17	Michael Gerndt	Handled comments of first review
0.3	22/08/17	Madhura Kumaraswamy	Handled comments of first review
1.0	22/08/17	Zakaria Bendifalla	Handled comments of first review
1.1	29/08/17	Michael Gerndt	Handled comments of second review
1.2	29/08/17	Zakaria Bendifalla	Handled comments of second review
1.3	29/08/17	Michael Gerndt	Final Version

Executive Summary

This deliverable specifies the READEX programming paradigm that allows the application expert to provide domain-level knowledge about the application dynamism to enhance the tuning process. READEX developed the Domain-level Knowledge Specification Interface (DKSI) that provides means to express the expert's domain knowledge related to the application structure, the application characteristics, and specific application-level tuning parameters. The specification is based on Score-P annotations and a novel library for specifying application-level tuning parameters.

The deliverable first presents the role of specification based on the formalization given in D4.1. It then introduces the concepts provided in each of the three areas by giving a short motivation, the way of specification and an example. The document closes with a description of the workflow for introducing domain knowledge into the READEX approach.

Contents

1 Introduction

While the READEX tool suite automates dynamic energy efficiency tuning, its result can be improved by the specification of domain knowledge through the application expert.

This deliverable specifies the READEX programming paradigm that allows the application expert to provide domain-level knowledge about the application dynamism to enhance the tuning process. READEX developed the Domain-level Knowledge Specification Interface (DKSI) that provides means to express the expert's domain knowledge related to the application structure, the application characteristics, and specific application-level tuning parameters.

The DKSI covers three aspects of domain knowledge:

1. Specification of application structure
2. Specification of application characteristics
3. Specification of application tuning parameters

Each of the three aspects will be introduced below in the context of the formalization given in Deliverable D4.1.

1.1 Application structure

The READEX methodology dynamically tunes *significant regions* $R_{sig} \subseteq R_{instr}$, a subset of all instrumented regions. Significant regions have to be coarse granular to be suitable for dynamic reconfiguration of tuning parameters. The structuring of the application into regions according to the programming language, e.g., subroutines, might not match the view of the developer of the application structure. Therefore, the DKSI offers means to specify additional regions manually.

READEX exploits different types of dynamism in applications. We distinguish *intra-phase dynamism* and *inter-phase dynamism*. In READEX we expect the application to have a phase region, which is one of the instrumented program regions R_{instr} and the major progress loop of the application. Each execution of the phase region is called a phase. Inter-phase dynamism means changes in the characteristics of the phases of the application. In contrast to inter-phase dynamism, intra-phase dynamism identifies different characteristics of significant regions within a single phase.

The application expert has to be involved in the identification of the phase region, since its selection has to take the overall architecture of the application into account.

1.2 Application characteristics

Optimal configurations for tuning parameters are not determined and switched for individual regions with different characteristics but on the granularity of individual *runtime situations* $rts \in RTS$. Each runtime situation is an execution of a significant region. *Identifiers* are used to distinguish runtime situations with different characteristics in the *Application Tuning Model (ATM)* generated by *Design Time Analysis (DTA).*

The ATM consists of a classification of rts' into *scenarios* $s \in S$. The *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. The rts context allows to distinguish different rts' of the same region based on *identifiers* and their value. Identifiers considered in READEX include *region name*, *region call path*, and *region identifiers*, *phase identifiers*, and *input identifiers.*

Region name and region call path are automatically detected identifiers for rts'. Additional identifiers can be given by the application expert. Region identifiers are used to distinguish rts' with different characteristics based on the execution context. Phase identifiers classify all rts' of entire phases based on phase characteristics. Input identifiers classify all rts' of an execution with respect to characteristics induced by the application input.

1.3 Application 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 READEX Runtime Library (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 $VAL = \cup_{tp \in TP} VAL_{tp}$.

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 $cfg : TP \longrightarrow VAL$.

READEX explores different types of tuning parameters, namely hardware parameters, system software parameters, and application parameters. While the first two are given by the execution environment, the application tuning parameters (ATP) are specific for the application. They can select different code paths or, for example, be special parameters to algorithms. The DKSI provides means for the expert to define such tuning parameters in the application code.

1.4 Structure of the deliverable

The deliverable presents the specification features of DKSI for the three areas. Section 2 covers the specification of regions and phase region, Section 3 presents the specification of identifiers, and Section 4 defines the DKSI features for ATPs. Section 5 explains the workflow for an application expert in using the DKSI to include domain knowledge into the analysis and tuning done by the READEX Tool Suite.

1.5 Example

Listing 1: Multigrid code structure

```
do it = 1, max_iter
    ...
    call VCycle(...)
    ...
enddo


subroutine VCycle(...)
//Proceed from finest to coarsest grid
do k = max_level, min_level+1
  ...
  call restrict(...)
enddo
...
//Propagate correction from coarsest to finest grid
do k = min_level+1, max_level
  call interpolate(...)
  call resid(...)
  call psinv(...)
enddo
...
end subroutine VCycle
```

In this document we use a multigrid solver as a running example. Listing 1 shows a high-level view of the MG (MultiGrid) benchmark of the NAS parallel benchmark suite [2]. MG uses a V-cycle to solve a discrete Poisson equation on a 3D grid. It is based on a hierarchy of grid levels, where the maximum level is the finest grid with the highest resolution.

During each iteration, an entire V-cycle is executed starting from the highest grid level. First, the result on the current grid level k is projected to the next coarser grid level $k-1$. When the coarsest grid is reached, an approximate solution is computed. The result is then interpolated from the coarser to the finer grid, where the residual is calculated and a smoother is applied to correct the result. The result is then propagated further upwards.

2 Specification of Application Structure

2.1 User regions

The decomposition of the program into program regions is typically defined by the syntax of the programming language. Common types of regions are subroutines, loops, and structured blocks. Besides standard regions, the application expert might be able to identify additional regions that are not represented as a standard region.

Typical usecases for user regions are:

- User regions can combine several calls to different functions which belong together and are too fine granular for switching the configuration individually.
- They can identify certain parts of an algorithm that would otherwise not be a target of tuning because this part is not represented by a standard region type of the programming language.
- If instrumentation is too fine granular and leads to a lot of overhead, automatic instrumentation can be switched off and significant regions can be manually instrumented.

2.1.1 Specification

User regions are defined in READEX with Score-P macros. These macros can enclose arbitrary code and are instrumented automatically. As a result, the regions can be handled by the READEX Tool Suite like any other region.

```
#include "SCOREP_User.inc"

SCOREP_USER_REGION_DEFINE(R1)

SCOREP_USER_REGION_BEGIN(R1, "name", SCOREP_USER_REGION_TYPE_COMMON)
  ...
SCOREP_USER_REGION_END(R1)
```

First the user region handle is defined with the macro `SCOREP_USER_REGION_DEFINE`. Then the start and end of the region are marked with the macros `SCOREP_USER_REGION_BEGIN` and `SCOREP_USER_REGION_END`. Both use the region handle, and the begin macro also specifies a name and a type, here the default region type. Details can be found in the Score-P User Guide [4].

2.1.2 Example

Listing 2 shows the multigrid example where all three function invocations used when the solution is propagated from the coarser to the finer grid are combined into a user region

`MGStep`. This enables DTA to tune this coarser region and avoids the overhead that would be introduced if each of the three subroutines were tuned individually.

Listing 2: The three functions executed on a given grid are combined into a coarse granular region called `MGstep`.

```
#include "SCOREP_User.inc"

SCOREP_USER_REGION_DEFINE(R1)

...

do k = min_level+1, max_level
  SCOREP_USER_REGION_BEGIN(R1, "MGstep", SCOREP_USER_REGION_TYPE_COMMON)
  call interpolate(...)
  call resid(...)
  call psinv(...)
  SCOREP_USER_REGION_END(R1)
enddo
```

2.2 Phase region

Central to the tuning approach of READEX is the concept of the phase region. Applications typically have a central progress loop that iteratively performs the computation. This loop can be implemented by a standard loop construct of the programming language, but it does not have to be implemented in this way. Even if it is a standard loop, it is not obvious how to figure out the phase region automatically. The code expert can easily provide this information. Therefore, READEX offers the user to specify the phase region as shown below.

Score-P offers the concept of an *online access phase region* to enable 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.

2.2.1 Specification

The specification of the phase region is very similar to a user region. The only difference is that names of the Score-P macros are `SCOREP_USER_OA_PHASE_BEGIN` and `SCOREP_USER_OA_PHASE_END`.

```
#include "SCOREP_User.inc"

SCOREP_USER_REGION_DEFINE(R1)

SCOREP_USER_OA_PHASE_BEGIN(R1, "name", SCOREP_USER_REGION_TYPE_COMMON)
  ...
SCOREP_USER_OA_PHASE_END(R1)
```

2.2.2 Example

Listing 3 shows the specification of the phase region for the MG benchmark. Several V-cycles are executed to solve the equation. Each cycle is marked as a phase and thus can be used to, for example, assess different candidate configurations during DTA.

Listing 3: The body of the iteration loop is marked as the phase region. Each iteration is a phase of the program.

```
#include "SCOREP_User.inc"

SCOREP_USER_REGION_DEFINE(R1)

do it = 1, max_iter
! phase region begins
  SCOREP_OA_PHASE_BEGIN(R1, "VCycle", SCOREP_USER_REGION_TYPE_COMMON)
    ...
    call VCycle(...)
    ...
  SCOREP_OA_PHASE_END(R1)
! phase region ends
enddo
```

3 Identification of Application Characteristics

3.1 Region identifiers

As well as the region name and the call path, additional region identifiers can be used to distinguish runtime situations with different characteristics. Without being able to identify and distinguish those runtime situations, the ATM cannot specify different configurations.

3.1.1 Specification

Region identifiers are specified as Score-P parameters for parameter-based profiling and can be of type integer and string as shown in the following listing. The parameters determine region identifiers for the compiler-instrumented region foo.

```
subroutine foo(integer myint, integer myuint)
  SCOREP_USER_PARAMETER_INT64("myint",myint)
  SCOREP_USER_PARAMETER_UINT64("myuint",myuint)
  // do something

end subroutine
```

3.1.2 Example

Listing 4: Specification of the grid level as region identifier.

```

subroutine VCycle(...)
...
!--- level k-1 to level k ---!
do  k = min_level+1, max_level
  call interpolate(..., k)
  call resid(...)
  call psinv(...)
enddo
end subroutine VCycle

!Interpolate to level k region
subroutine interpolate(..., k)
  SCOREP_USER_PARAMETER("level",k)
  ...
end subroutine interpolate
```

In the MG benchmark, the size of the grid processed in `interpolate(..., k)` in Listing 4 gets higher when going from the minimum grid level to the maximum. At a certain grid level, the computation switches from being compute bound to memory bound. To enable DTA to determine special system configurations for compute and memory bound runtime situations, the application expert can add a region identifier for the grid level inside the `interpolate` region (Line 14 of Listing 4). In contrast to the calling context tree (CCT) without region identifiers in Figure 1, the grid level can be used as an identifier to distinguish the runtime situations of the region `interpolate` [3], as shown in Figure 2. The tool suite cannot detect and distinguish the runtime situations of `interpolate` automatically, and hence, it is the responsibility of the application expert to provide hints to do so.

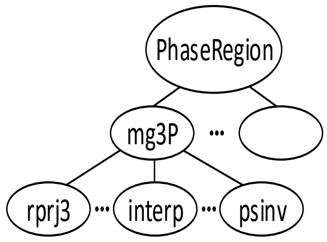

Figure 1: Calling Context Tree (CCT) of MG without region identifiers.

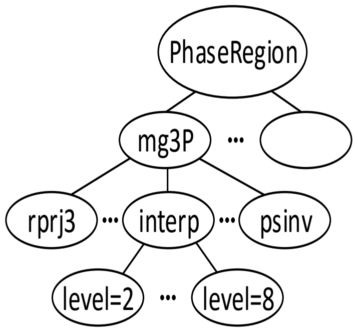

Figure 2: CCT of MG with region identifiers.

3.2 Phase identifiers

Phase identifiers identify phases with different characteristics. They can be used in the ATM to distinguish rts' based on the different behavior of the phase. Thus, different configurations can be given for rts' in these different phases.

3.2.1 Specification

Phase identifiers are provided in the same way as region identifiers via Score-P parameters that are attached to the phase region. Phase identifiers, such as the degree of sparsity of the matrix or the arithmetic intensity of the phase region, must enable the prediction or selection of different best configurations for phases that have varying characteristics or behavior.

3.2.2 Example

INDEED [1] is a sheet metal forming simulation software with an implicit time integration. It uses an adaptive mesh refinement, and the number of finite element nodes that it uses increases with every time step, resulting in an increasing computational cost. Figure 3 shows the workflow of the INDEED application. There are three loops, i.e., the time loop, the

Table 1: Best configurations for INDEED clusters

Cluster	Tuning Parameters		Normalized Energy	Energy
	CPU_FREQ	UNCORE_FREQ		
1	1.5 GHz	1.8 GHz	0.00342527	1872 J
2	2.4 GHz	1.4 GHz	0.00246028	3145 J
3	1.8 GHz	1.2 GHz	0.00146470	6074 J

contact loop and the equilibrium loop, that have varying levels of dynamism. The application expert may annotate any of these loops as the phase region.

In this example, the time loop was used as the phase region. Figure 4 indicates that the first phases are computationally very cheap. This is due to the fact that there is no contact between tools and workpiece yet. On initial contact, a lot of computational work is required, and we obtain a peak in the graph. The following peaks then arise due to more refined regions of the tool making contact with the workpiece.

The phase identifiers *Compute Intensity* and *Branch Instructions* enable the DTA to cluster the phases having different behavior into different groups, as shown in Figure 5. To group the phases having similar characteristics, the phase identifiers are normalized. In this example, three clusters are obtained, and the unclustered data points are labeled as noise. The best configuration for each cluster is obtained for the least value of energy normalized by the total number of instructions. Table 1 shows the best configuration for each cluster.

3.3 Input identifiers

The READEX tool suite will not only tune the application for a single input but will learn from running the application for different inputs. Input identifiers allow to characterize the variations among executions with different sets of application inputs. As a result, DTA will be able to identify more rts's with different characteristics which will eventually improve the tuning model. Input identifiers may be simple, such as the grid size of the application domain and the number of processors, or complex, like the number of contact points in metal forming simulations.

3.3.1 Specification

The DKSI allows to provide input identifiers in an accompanying input specification file in the form of key-value pairs as given in Listing 5.

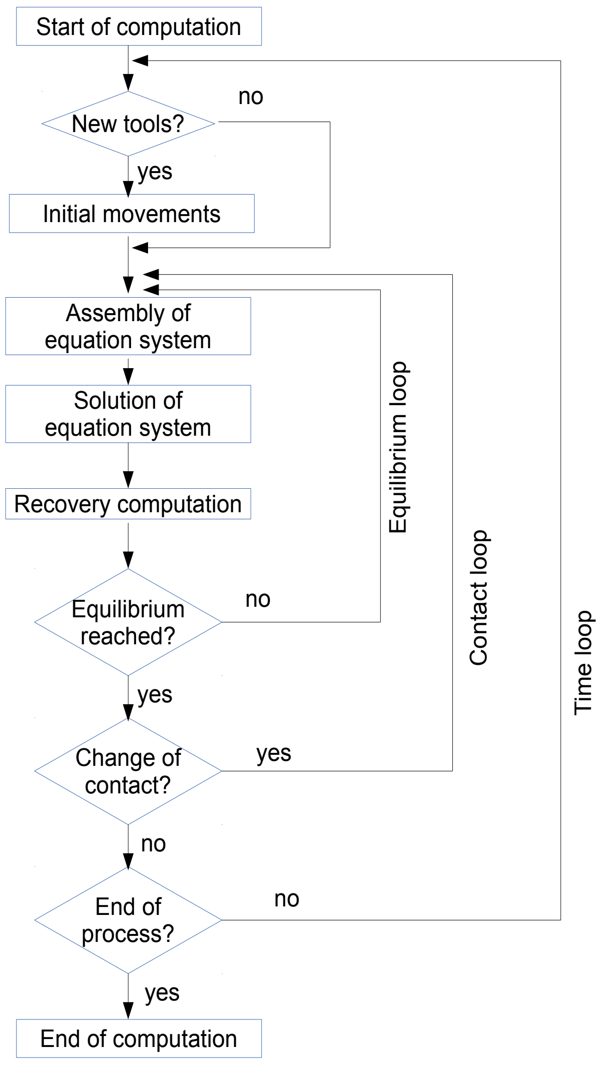

Figure 3: Loops in INDEED that are candidates for the phase region.

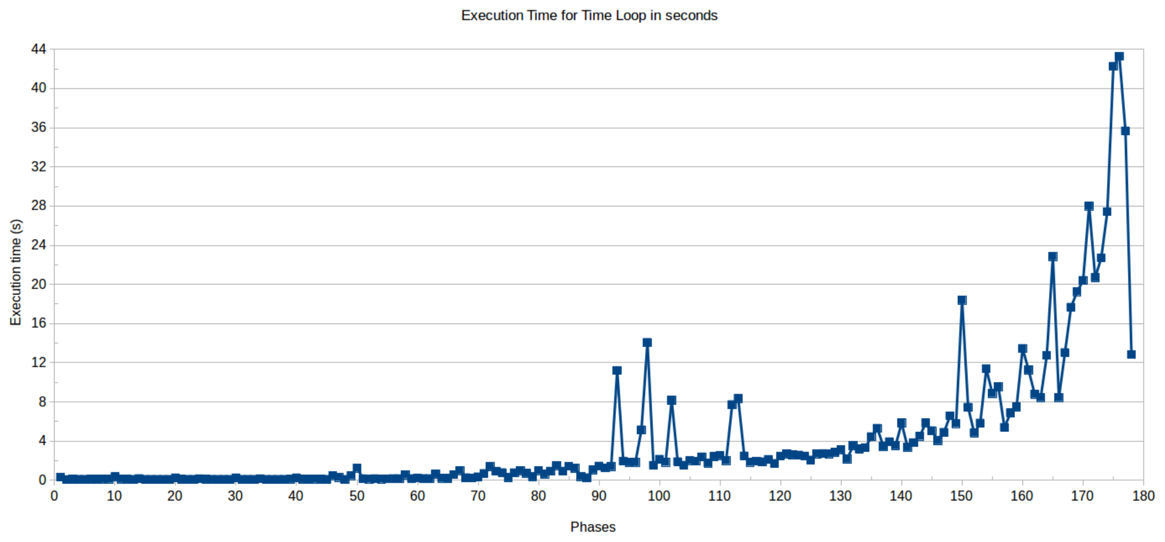

Figure 4: Execution time vs. phases of the time loop in INDEED.

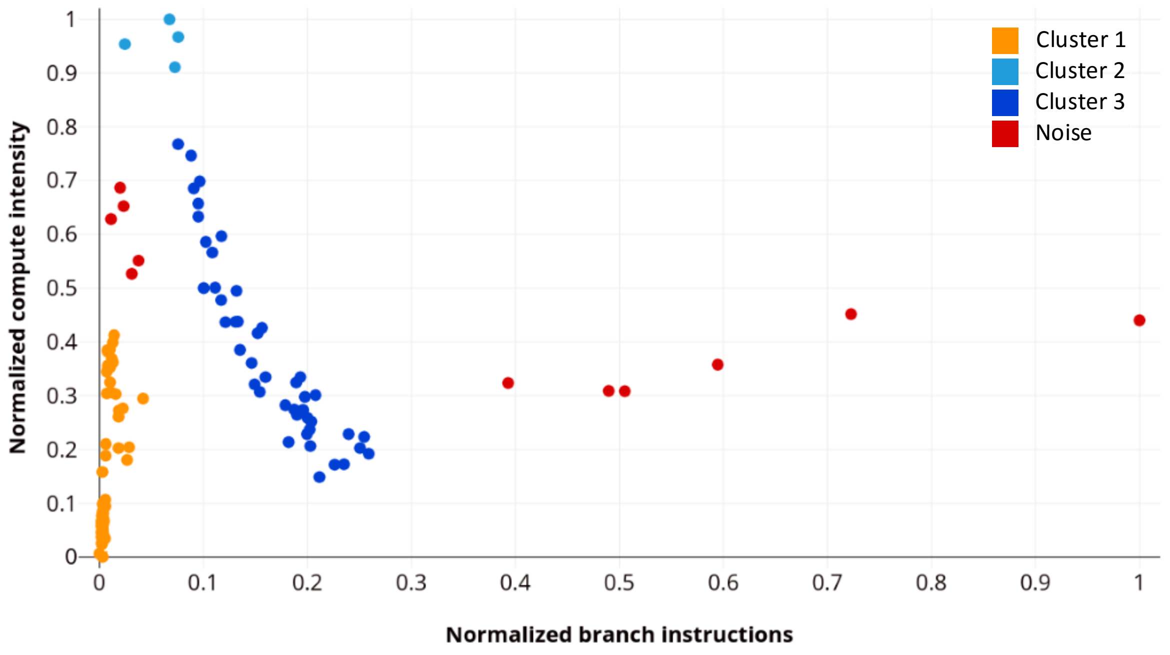

Figure 5: Clusters obtained for normalized compute intensity vs. normalized branch instructions in INDEED.

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

Listing 5: Input identifier specification file IID_SPEC

```
export INPUT_IDENTIFIER_SPEC_FILE="IID_SPEC"
```

Listing 6: Specify input identifier file in RRL

Input identifier specification files are used by both PTF and the RRL. They are passed to PTF via the command line using the flag `--input-desc="IID_SPEC"`, and to the RRL via an environment variable, `INPUT_IDENTIFIER_SPEC_FILE` shown in Listing 6.

It is not necessary to pass the same input identifier specification file to both PTF and the RRL. If input identifiers given for one input file are missing for another input file, the missing identifiers are handled in DTA with a default value.

3.3.2 Example

In the MG benchmark, the size of the finest grid determines at which grid level the application switches from memory bound to compute bound. Thus, the region identifier giving the grid level for the rts' of the functions applied on a grid level is not sufficient for a general tuning model covering varying input sizes. The optimal configuration is also dependent on the size of the finest grid and thus this grid size has to be taken into account when selecting the configuration for a certain grid level at runtime. In addition to the size of the finest grid also the number of processes is important. The more processes are used, the better the data distribute over the caches and the earlier, in terms of grid level, the application switches between memory and compute bound. The numbers of processes and threads are considered as standard input identifiers and need not be given in the specification file.

4 Application Tuning Parameters

In addition to hardware and system-software tuning parameters, READEX also targets application level parameters, that is, parts of the code itself which could be used as tuning parameters. The simplest example of this is the case where different implementations of the same algorithm are available, each having its own impact on performance and energy. The aim of using application level parameters is to exploit the possibility to switch between the different implementations or, in a more general term, the possibility for READEX to choose between code level alternatives.

The three main steps that drive the handling of ATP are: 1) source code annotation, 2) generation of an ATP description file and tuning in the DTA via the READEX plugin, and 3) use of the DTA results during RAT.

The annotation step (Figure 6) is primarily done by the application developer and consists of first, instrumenting the program code to prepare control variables, these are the program variables used to tweak programs semantics, such as choosing between different code paths or defining different blocking factors, at second, comes the annotation part where API calls mark the control variables and describe their types, ranges and possible dependences between them, thus providing a READEX tuning system with the necessary information to tune the variables. READEX provides the necessary API through the *ATP library* (*ATPlib*). Once the annotation is finished the application is compiled to produce the executable which will be used in the DTA phase.

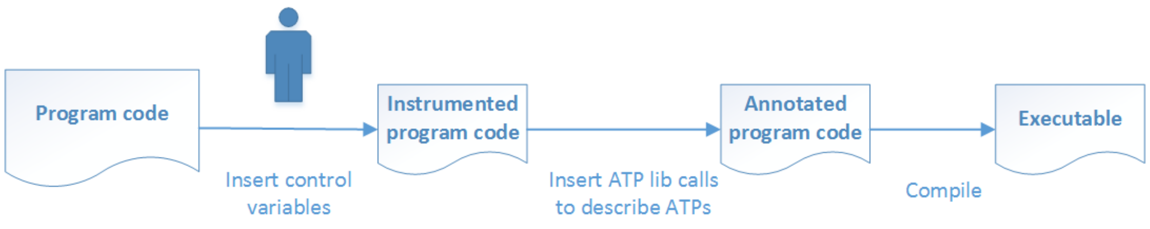

Figure 6: ATP declaration through application instrumentation and annotation.

In DTA (Figure 7), the first phase of the application execution is reserved to the discovery of the details about the ATPs, their ranges, constraints and location. The information found is recorded into an ATP description file. READEX then uses the remaining phases of the application to trigger tuning actions which modify ATPs as it is done for hardware and system-software tuning parameters. The best found configurations for ATPs are stored in the same tuning model file as for the other parameters.

The third step of the READEX methodology uses the previously produced Tuning Model to dynamically tune the application by setting control variables with the best found values in the DTA (Figure 8). However, since the handling of ATPs involves manual code instrumentation

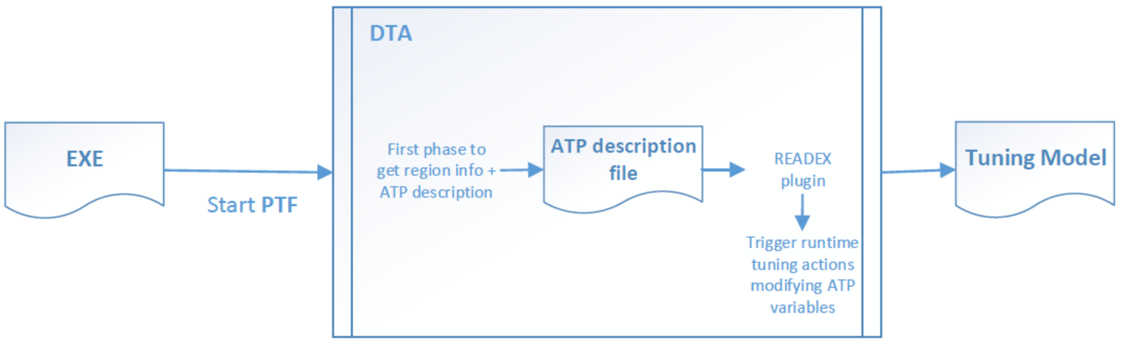
DTA
EXE
Start PTF
First phase to get region info + ATP description
ATP description file
READEX plugin
Trigger runtime tuning actions modifying ATP variables
Tuning Model

Figure 7: ATP handling during DTA.

by the developer, some of the API calls inserted and used in the DTA phase, such as parameter declaration functions, will be deactivated in the RAT phase.

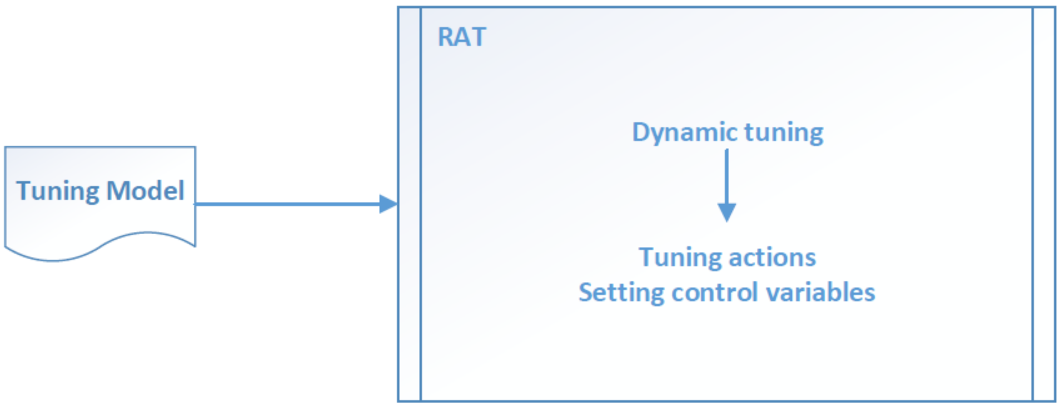

Figure 8: ATP handling during RAT.

The correct declaration and description of ATPs by the user is key to the success of the process. This goes by defining mainly the *types* of variables accepted by the tuning system, their *ranges* and the dependences between them if any. These are detailed in the following sections.

4.1 ATP declaration and retrieval

The control variables used for ATPs can be of diverse types. This includes simple data structures such as integers, booleans or floats as well as complex types such as arrays or structures.

The current version of the READEX tool suite is set to handle *integer* data types, as exploring integer ranges and solving constraints between them requires a relatively reasonable amount of compute power compared to floats and complex structures.

In the cases where a tuning parameter is a complex structure, the user may, when possible, simplify it by mapping its possible values to integer variables.

Two ways of expressing ATP values are made possible. These are: 1) in the form of a *range* of values by giving an array of three values (minimum value, maximum value and an increment), and 2) in the form of an *enumeration* of values by giving an array of all the possible values the parameter may take.

Also, application tuning parameters differ a little from other hardware and runtime parameters in that they cannot be controlled at each region entry and exit. Their values can only be taken into account when the corresponding part of code is being executed.

4.1.1 Specification

Two API functions are used for the declaration of a new application tuning parameter. The first is used to declare parameter name, type, default value and domain. The second allows to supply the possible values of the parameter. These are provided in Listing 7.

The overall handling of ATP values follows the same process as any other tuning parameter in READEX. The tuning process is driven by PTF, and parameter values are extracted by the RRL; the latter sends values to control plugins. However, the difference lies in the last layer, where ATPs do not have a control plugin, instead an API call assigns the value to a code variable. At the API level, this is translated through the third API call.

```
ATP_PARAM_DECLARE(const char *param_name, const char param_type, int
    default_value, const char *domain_name)
ATP_PARAM_ADD_VALUES(const char *param_name, void *vArray, int array_size,
    const char *domain_name)
ATP_PARAM_GET(const char *param_name, void *tp_address, const char *domain_name)
```

Listing 7: API function calls for ATP declaration.

The other concept introduced with the current specification is the notion of *domain*. The concept will be detailed further in Section 4.2.

4.1.2 Example

In Listing 8, API calls are used to declare the `atp_cv` variable to READEX as an application tuning parameter and also to extract its values from the tuning system. In DTA, READEX explores different values and assesses their performance, and in RAT only the best found values are assigned.

```
void foo(){
  int atp_cv;

  ...
  ATP_PARAM_DECLARE("solver", RANGE, 1, "DOM1");
  ATP_ADD_VALUES("solver", {1,5,1}, 3, "DOM1");
  ATP_PARAM_GET("solver", &atp_cv, "DOM1");

  switch (atp_cv){
    case 1:
        // choose algorithm 1
        break;
    case 2:
        // choose algorithm 2
        break;
    ...
  }
```

Listing 8: Example of ATP declaration and value retrieval into the control variable `atp_cv` through ATPlib API functions.

4.2 Domains

An application source code may contain several application level parameters. Ideally, these would be independent from each other. In practice, this is not necessarily the case; the values of a parameter may depend on those of another one. This would translate into the notion of constraints between parameters (detailed in Section 4.3). Therefore, the application developer may indicate to READEX that a number of parameters have constraints between them by putting them in the same domain.

4.2.1 Specification

In the API, domains are declared within the declaration of a parameter in the same call, where both parameter name and domain name need to be supplied. Also, if no domain is declared at a parameter declaration then the "default" domain is assigned to it. Listing 9 recalls the API call to declare a tuning parameter and illustrates that this same call also serves as the domain declaration.

```
ATP_PARAM_DECLARE(const char *param_name, const char param_type, int
    default_value, const char *domain_name)
```

Listing 9: API function calls for ATP declaration.

4.2.2 Example

Listing 8 provides an example of domain assignment where the declared variable `solver` is set to belong to the `DOM1` domain.

4.3 Constraints

As mentioned earlier, in the case where two or more ATPs have constraints between them, it is necessary to express the constraints to READEX, as this would allow to generate only valid values for the tuple of parameters.

In order to take dependencies between ATPs into account, the ATP description formalism allows these dependencies to be expressed mathematically in the form of constraints. An example of such constraints is:

```
Let mesh, solver be integers
solver ranges in [1,5] with an increment of 1
mesh   ranges in [0,80] with an increment of 10
constraints:
1. if solver equals one them mesh must be between 0 and 40.
2. if solver equals two them mesh must be between 40 and 80.
3. if solver is above two them mesh must be equal to 120.
```

As the listing shows ATPs solver and mesh are tied by the constraints 1,2 and 3. The logical expression form that READEX accepts is:

```
(solver = 1 && 0 <= mesh <= 40)|| (solver = 2 && 40 <= mesh <= 80)||
(solver > 2 && mesh = 120)
```

In a nutshell the logical expressions accepted to describe constraints are those built using the following rules and operators:

1. Addition, Subtraction, Multiplication and Division (+ , - , * , /) are accepted to form affine functions.
2. The following operators are accepted to connect affine functions and construct logical expressions: $<, >, =, <=, >=, ! =$.
3. The following operators to connect logical expressions are accepted: &&, ||.

4. It is also possible to use parenthesis to override the base operators precedence : (,).

In order to avoid increased complexity in constraint solving, READEX accepts affine function based constraints only. These are sufficient to handle a wide range of constraints.

4.3.1 Specification

Constraint declaration goes through a single API function call as illustrated in Listing 10, where the constraint is given in the form of a logical expression.

```
ATP_CONSTRAINT_DECLARE(const char *constraint_name, const char
    *constraint_expr, const char *domain_name)
```

Listing 10: API call for constraint declaration through ATPlib API functions.

4.3.2 Example

An example of the declaration of a constraint between two ATPs is illustrated in Listing 11, the concerned variables in the example are `solver` and `mesh`.

```
void bar(){
  int atp_ms;

  ...
  ATP_PARAM_DECLARE("mesh", RANGE, 40, "DOM1");
  ATP_ADD_VALUES("mesh", {0,80,10}, 3, "DOM1");
  ATP_CONSTRAINT_DECLARE("const1", "(solver = 1 && 0 <= mesh <= 40)||
                                  (solver = 2 && 40 <= mesh <= 80)||
                                  (solver > 2 && mesh = 120)", "DOM1");
  ATP_PARAM_GET("mesh", &atp_ms, "DOM1");

  if(atp_ms > 1 && atp_ms <= 40)
  {
      ...
  }
  if(atp_ms > 40 && atp_ms <= 80)
  {
      ...
  }
  if(atp_ms == 120)
  {
      ...
  }
  ...
}
```

Listing 11: Example of ATP constraint declaration through ATPlib API functions.

4.4 Exploration

Tuning parameters may have a big number of possible values, which can present a difficulty during the tuning phase to explore all the values. Therefore, the tuning system makes use of heuristics to guide the exploration and minimize its time cost. However, as the user can be knowledgeable of the details of the tuning parameters he declared, he can give hints to the tuning system about what heuristics would be better to use in the exploration.

4.4.1 Specification

The ATP library API provides a function call for the programmer to give hints to the READEX tuning system about what heuristics to use. The call allows to give an ordered list of exploration heuristics to the READEX system. It should be noted that the hints are tied to a domain which means that all the parameters included in the domain would be subject to the same hints. Listing 12 illustrates the exploration hints functions call.

```
ATP_EXPLORATION_DECLARE(const char *explorations_list, const char *domain_name)
```

Listing 12: API function calls for ATP declaration.

4.4.2 Example

An example of use of the exploration API call is given in Listing 13.

4.5 API code organization

The ATP library API offers a bit of flexibility regarding where to declare the parameters. However, for good readability and proper functioning of the tuning system, some rules and subtleties need to be taken into account. These are:

- The API call for declaring a tuning variable must be called before declaring its values which also need to be called before the call of value retrieval. This means that the functions for parameter declaration, value declaration and value retrieval must be all called before the variable to which the parameter is tied is used to make the decision.
- Constraint API calls can be put anywhere, even if the tuning variables referenced in its logical expression have not been declared yet.
- Exploration API calls can be put anywhere in the code.

- The common condition for all API calls is that the calls must be reached by the execution flow during the first phase of the application.
- For the readability of the code it is better to have:
  - Tuning variable declaration close to the corresponding control variable declaration.
  - Tuning variable value retrieval call just before the decision statement involving the corresponding control variable.
  - Constraint declaration when all tuning variables involved in its expression have been declared.

4.6 API Example

Listing 13 illustrates how ATPlib API functions can be used to declare variables from the code as tuning parameters. It also shows how constraints between these variables can be declared as well.

```
void foo(){
  int atp_cv;

  ...
  ATP_PARAM_DECLARE("solver", RANGE, 1, "DOM1");
  ATP_ADD_VALUES("solver", {1,5,1}, 3, "DOM1");
  ATP_PARAM_GET("solver", &atp_cv, "DOM1");

  switch (atp_cv){{
    case 1:
        // choose algorithm 1
        break;
    case 2:
        // choose algorithm 2
        break;
    ...
  }

  int32_t hint_array = {GENETIC, RANDOM};
  ATP_EXPLORATION_DECLARE(hint_array, "DOM1");
}

void bar(){
  int atp_ms;

  ...
  ATP_PARAM_DECLARE("mesh", RANGE, 40, "DOM1");
  ATP_ADD_VALUES("mesh", {0,80,10}, 3, "DOM1");
  ATP_CONSTRAINT_DECLARE("const1", "(solver = 1 && 0 <= mesh <= 40)||
                                 (solver = 2 && 40 <= mesh <= 80)||
                                 (solver > 2 && mesh = 120)", "DOM1");
  ATP_PARAM_GET("mesh", &atp_ms, "DOM1");

  if(atp_ms > 1 && atp_ms <= 40)
  {
      ...
  }
  if(atp_ms > 40 && atp_ms <= 80)
  {
      ...
  }
  if(atp_ms == 120)
  {
      ...
  }
  ...

}
```

Listing 13: Example of ATP exploitation through the ATPlib API functions.

5 Domain Knowledge Specification Workflow

Figure 5 presents the integration of the specification of domain knowledge into the overall DTA workflow.

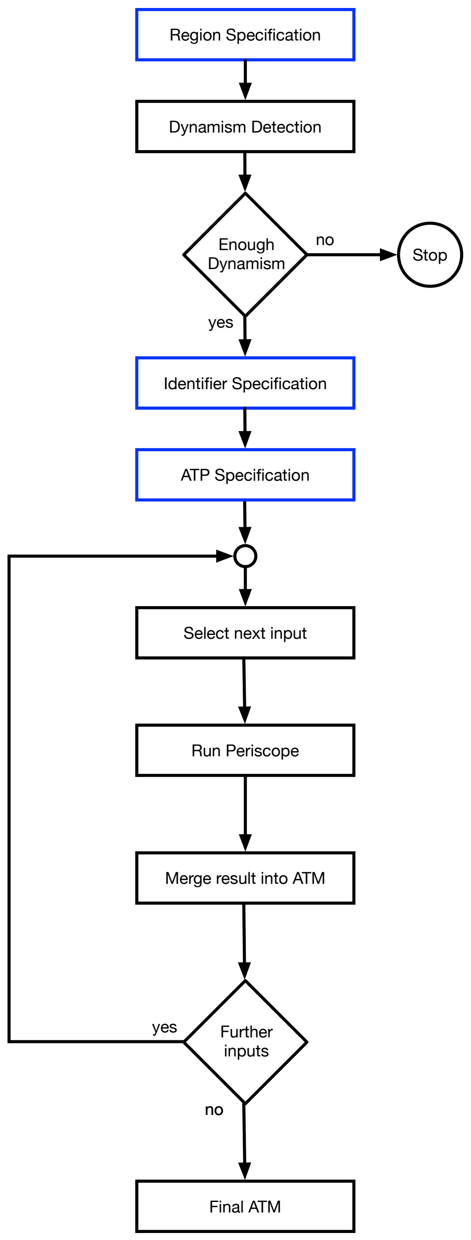

Figure 5: Integration of domain knowledge specification into the DTA workflow.

Before application dynamism is detected with the help of `readex-dyn-detect`, the application expert specifies the phase region and may specify additional program regions.

Based on the result of dynamism detection, either READEX tuning is stopped in case of no available dynamism, or the application expert may add additional identifiers to support DTA. These identifiers are region identifiers, phase identifiers, and input identifiers. The identifiers will allow DTA to generate a more sophisticated ATM. While region and phase identifiers are specified in the source code via Score-P parameters, input identifiers are added in files accompanying the original input file.

The final step in preparing DTA is to specify ATPs in the application source files via calls to the ATP library. When the application is executed, the ATPs are written to the ATP description file which is then read by Periscope at the start of the analysis.

DTA then proceeds analyzing all the given inputs. For each input an ATM is generated and merged into the final ATM that is then given to RAT. DTA terminates after all inputs were processed.

The application expert might add some additional manual loops to the basic control flow. The generated tuning results and the ATM might allow the expert to add additional DKSI specifications in order to further enhance the tuning result. Depending on the type of additional specification, the steps following the specification step in the control flow will have to be executed again.

6 Summary

This deliverable presented the specification means of DKSI that allow the application expert to guide DTA to improve the result beyond the automatic version. It is the implementation of the new READEX programming paradigm.

User regions can be used to aggregate too fine granular regions to enable their tuning. The specification of the phase region allows to explore inter-phase dynamism in addition to the intra-phase dynamism of the application. Region identifiers name different characteristics of a computation and can be used to enhance the tuning model by distinguishing runtime situations. Input identifiers characterize different input behavior and enable DTA to generate a more global tuning model. DKSI also enables new tuning parameters specified via the ATP library. DTA then takes these tuning parameters into account via a more extensive search space for energy tuning. The implementation support for DKSI is presented in further deliverables.

References

[1] INDEED – Highly Accurate Finite Element Simulation for Sheet Metal Forming. `http://gns-mbh.com/products/indeed`. Last accessed October 14, 2015.

[2] David H. Bailey. The NAS Parallel Benchmarks. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1254–1259. Springer, New York, 2011.

[3] Anamika Chowdhury, Madhura Kumaraswamy, Michael Gerndt, Zakaria Bendifallah, Othman Bouizi, Lubomír Říha, Ondřej Vysocký, Martin Beseda, and Jan Zapletal. Domain knowledge specification for energy tuning. In *2nd Workshop on Power-Aware Computing 2017*, June 2017.

[4] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf. Score-p: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In H. Brunst, M. Müller, W. E. Nagel, and M. M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91. Springer, Berlin, 2012.