Energy Aware Runtime (EAR): Internals document

This document is part of the Energy Aware Runtime (EAR) framework. It has been created in the context of the BSC-Lenovo Cooperation project.

Contact

BSC: Julita Corbalan julita.corbalan@bsc.es

Lenovo: Luigi Brochard lbrochard@lenovo.com

NOTE: This document describes internals for the Energy Aware Runtime (EAR). The “Energy Aware Runtime(EAR): reference” document must be read before.

Energy Aware Runtime (EAR) internals

1 Components included with EAR distribution

EAR distribution includes two core components: EAR library (***ear_lib***) and EAR daemon (**ear_daemon**).

ear_lib component is a runtime library (libEAR.so) dynamically loaded next to the application. Its components manages the following steps:

- MPI dynamic interception
- Automatic detection of application iterative structure using DynAIS
- Implement the EAR lifecycle (EAR state diagram)
- Collects those metrics that do not require privileged capabilities
- Ask EAR daemon for those metrics that requires privileged capabilities
- Compute performance/power projection
- Enforce the energy policy and asking the daemon to change the node frequency (if needed)
- Report performance metrics

ear_daemon is an external process executed next to applications in same nodes. It implements those services that requires privileged capabilities, such as changing the node frequency or reading uncore counters to compute memory bandwidth.

Although the libEAR.so is loaded with all the MPI, the EAR functionality is only executed in the context of one process per node. In the current version, EAR assumes MPI processes are equally distributed by nodes. Figure 1 shows an example of an MPI job with four processes executed in two nodes.

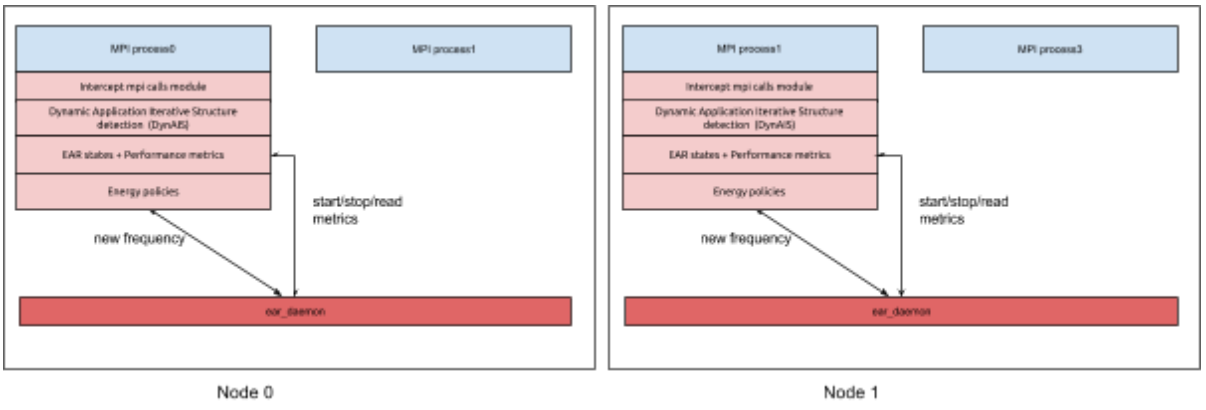

Figure 1: EAR executed with mpi processes in two nodes

Apart from *ear_lib* and *ear_daemon* components, EAR distribution also includes additional components:

- *ear_slurm_plugin***.** It implements a SLURM Spank plugin that integrates EAR utilization in the SLURM commands, hiding implementation details to users.

- *ear_utils*: A set of commands to execute the learning phase automatically in N nodes and to show, in text mode, the content of a EAR system database file.
- *ear_learning_phase*: files (except kernels) to automatically execute the learning phase. Includes scripts to start it in N nodes, to execute kernels in the N nodes and the computation of hardware signature.
- *ear_learning_phase/*kernels: The source code of kernels used in the learning phase.
- *ear_scripts*: a set of scripts to execute EAR remotely.
- *ear_scripts/tests*: easy examples of EAR utilization. This folder includes scripts to be submitted with SLURM and based on ssh command. This second option requires root privileges to start the *ear_damon*.

2 EAR library

Figure 2 shows the the EAR stack for a single process. Each subfolder in the *ear_lib* components implements features of one levels. We will describe main files and functionality.

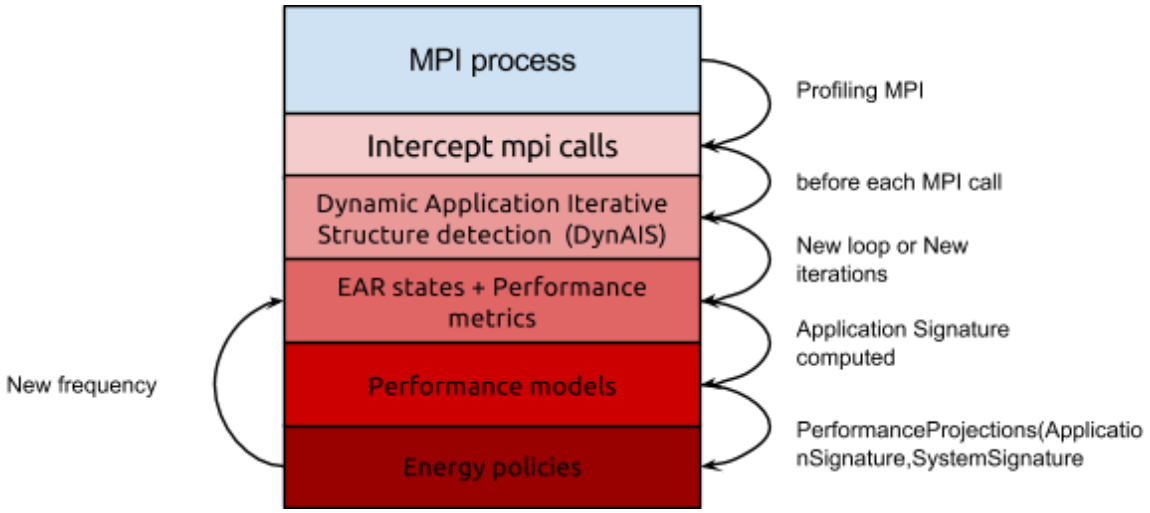

Figure 2: EAR stack

- Intercept MPI calls: *ear_dyn_inst*
- Dynamic Application Iterative Structure detection (DynAIS): *ear_dynais*
- EAR lifecycle management: *ear_states*
- Performance metrics: *ear_metrics*
- Performance/power models and Energy policies: *ear_models*
- *ear_gui/ear_frequency/ear_db* implements services used by other parts of EAR.

Figure 3 shows the main interactions between EAR files, reflecting also the EAR library behavior. The MPI interposition files calls DynAIS API to detect application outer loops. Once detected, *ear_base* functions that drives EAR behavior are called.

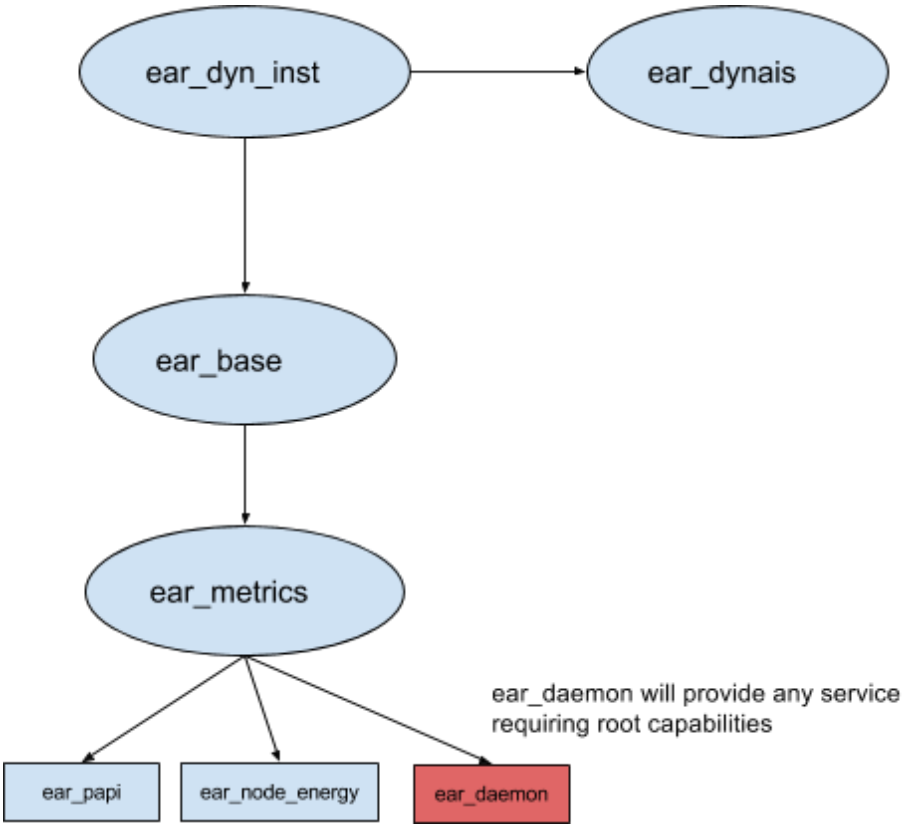

Figure 3: Main EAR files

2.1 Intercepting MPI calls

The profiling MPI interface defines two types of symbols: weak and strong symbols. The MPI interface defines weak symbols whereas the MPI library implements both. These weak symbols can be dynamically replaced by another set of symbols by defining the LD_PRELOAD environment variable. This mechanisms allows adding functionality before and/or after any MPI call, or even replacing the MPI call. The interposition files can be found in the ear_dyn_inst folder.

Code1 shows the code EAR provides to substitute the MPI_Allreduce function. We insert two EAR calls (before and after) the PMPI_Allreduce strong symbol. A similar approach is applied to all the MPI functions. The weak MPI symbols can be found in ear_dyn_inst/MPI_intercept.c.

```
int MPI_Allreduce(MPI3_CONST void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
MPI_Op op, MPI_Comm comm) {
   EAR_MPI_Allreduce_enter(sendbuf, recvbuf, count, datatype, op, comm);
   int res = PMPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm);
   EAR_MPI_Allreduce_leave();
   return res;
}
```

Code1: MPI interception wraps all the MPI calls and adds calls to EAR

All the entry/exit points are normalized to two functions: before_mpi and after_mpi. Next figure shows the corresponding code for the previous example. Except for MPI_init and MPI_finalize, EAR uses the same approach. All EAR functions used to normalize for many "EAR_…" call to before_mpi function can be found at ear_dyn_inst/MPI_interface.c file. (see code2)

```
void EAR_MPI_Allreduce_enter(MPI3_CONST void *sendbuf, void *recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, MPI_Comm comm) {
   before_mpi(Allreduce, (p2i)sendbuf,(p2i)recvbuf);
}

void EAR_MPI_Allreduce_leave(void) {
   after_mpi(Allreduce);
}
```

Code2: MPI interface normalizes all the MPI calls to before_mpi and after_mpi

The first argument in blue for the before_mpi function is used to classify the "type" of MPI call. The ear_dyn_inst/MPI_calls_coded.h file includes the EAR classification for all MPI calls. Since not all MPI calls has the same number and type of arguments, we have selected the most representative for each call, selecting three of them for each one (using null values when the MPI call has less than three arguments).

Code3 shows before_mpi function (implemented in processMPI.c). It creates a dynamic event based on its arguments and calls dynais (see DynAIS API in next section). DynAIS process the input and returns the status of the algorithm, reporting whether this event is part of a loop or not, and, in case it belongs to a loop is identified as the "first" element in the sequence (indicating a new loop or new iteration). Next section describes DynAIS internals with more detail.

Functions states_begin_period/states_end_period/states_new_iteration (implemented at ear_base/ear_base.c) are called when DynAIS reports these special states.


```
void before_mpi(mpi_call call_type, p2i buf, p2i dest)
{
   int ear_status;
   unsigned long ear_event = create_event(buf,dest,call_type);
   ear_status=dynais(ear_event,&ear_size,&ear_level);
   switch (ear_status){ // EAR lifecycle is driven by DynAIS returned state
         case NO_LOOP:
         case IN_LOOP:
                  break;
         case NEW_LOOP:
                  iterations=0;
                  states_begin_period(ear_event,ear_size);
                  ear_loop_size=ear_size;
                   break;
         case END_NEW_LOOP:
                  states_end_period(ear_loop_size,iterations,ear_event);
                  iterations=0;
                  ear_loop_size=ear_size;
                  states_begin_period(ear_event,ear_size);
                  break;
         case NEW_ITERATION:
                  iterations++;
                  states_new_iteration(ear_loop_size,iterations,ear_event,ear_level);
                  break;
         case END_LOOP:
                  states_end_period(ear_loop_size,iterations,ear_event);
                   iterations=0;
                  break;
         default:
                   break;
   }
```

Code3: DynAIS states drives EAR

2.2 Dynamic Application Iterative Structure detection: DynAIS

DynAIS offers the basic infrastructure to detect application iterative structures and allows the client (EAR in this case), to add any additional information such as: iteration time or specific metrics.

In this way, DynAIS provides a basic mechanism easily extended with any required functionality or metric. The success of DynAIS in detecting the application structure depends on the ability of generating event identifiers during the application execution. It is implemented as a recursive algorithm that detects nested loops up to a maximum user defined level.

The output of DynAIS is: BEGIN_NEW_LOOP if the event corresponds to the first event of a newly detected loop; BEGIN_NEW_ITERATION if the event if the first one in a new iteration of al previously detected loop; If the event is not included in the iterative sequence of events, it crashes the loop, END_LOOP is returned; If the event belongs to a loop structure but it is

not the first one in the iterative sequence, IN_LOOP is returned; NO_LOOP is returned in case the event does not belong to a loop or DynAIS has not yet detected it.

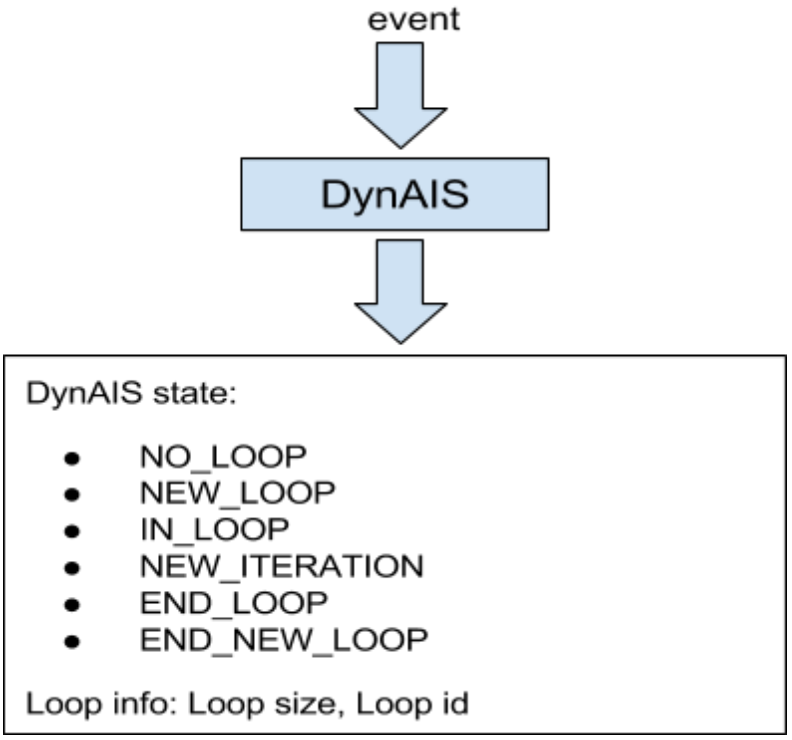

Figure 4: DynAIS states

DynAIS is implemented based on two technologies: A basic algorithm that detects simple repetitive sequences of discrete events and, a complex multi-level able to detect nested repetitive sequences (loops with loops inside) up to N levels. In this document, when we refer to DynAIS, it means Multi-Level DynAIS (see figure 5). Otherwise we'll say basic DynAIS.

Basic DynAIS uses concepts that comes from the context of signal processing [REF]. We have adapted these ideas to the specific case of using discrete values, where two values are equal only when the difference is 0. Basic DynAIS has been updated to consider, rather than only simple events in the input, events composed of two values: (ID, size). In the case of EAR, (ID,size) means (LoopID,LoopSize). This main change allows DynAIS to be instantiated recursively, passing the as "event" from one level to the next one the two elements that identifies a loop.

Multi-Level DynAIS is implemented as a recursive algorithm, where each level calls its upper level (level+1, starting from level 0, up to N, defined in the algorithm initialization function), when a new loop is reported. In order to use the same Basic DynAIS algorithm, the multi-level algorithm adapts the input of level 0 to (event_ID,1). Figure 5 shows the main steps of our algorithm. The first level processes all the events. Each time one level (starting at level 0) detects a new loop, the loop id and the loop size for this new loop becomes the input of the next level (LoopID,LoopSize). To compute the output, multi-level algorithm takes into account the status reported in the upper levels.

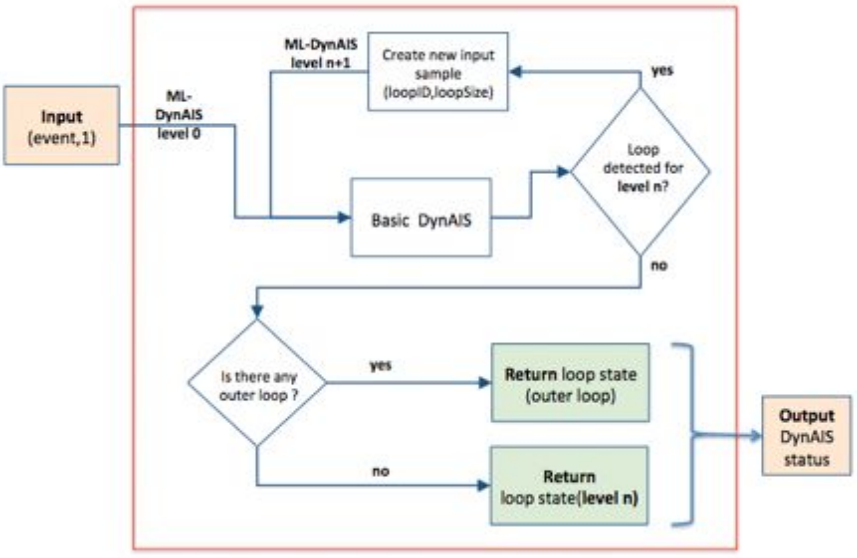

Figure 5: DynAIS multi-level algorithm

2.2.1 DynAIS API

Table 1 describes the DynAIS algorithm API. It is very simple since DynAIS hides the complexity of the multi-level algorithm.

Function (ear_dynais/ear_dynais.h)	Description
int dynais_init(unsigned int size, unsigned int levels);	Initializes DynAIS functionality with as many as specified in levels argument each one with size buffers size. Its buffer size limits the number of events in one level. For instance: ● size=100, levels=1, DynAIS will detect loops with 100 events per iteration as maximum ● size=100, levels=2, DynAIS will detect loops with 100 inner loops with size 100 as maximum (nested loops) Return 0 on success and -1 on error
int dynais(unsigned long event, unsigned int *size, unsigned int *level);	Receives an event as input and returns the DynAIS status. In case the event corresponds with the first event of an iteration, it updates size and level with the loop size (including nested levels) and the number of nested loops detected. Return one of the following DynAIS states (defined at ear_dynais.h): ● END_LOOP

	• NO_LOOP • IN_LOOP • NEW_ITERATION • NEW_LOOP • END_NEW_LOOP
void dynais_dispose();	Release all memory reserved at dynais_init.

Table 1: DynAIS API

2.2.2 DynAIS states

- NO_LOOP: The event is not an event of previously detected loop.
- NEW_LOOP: The event is identified by DynAIS as the “first” element in a loop, and it is the first iteration. DynAIS needs at least two iterations to identify a sequence of events is iterative, so, the event that will generate the NEW_LOOP event is not, in practice, the first one in the loop, but it’s the “first” identified by DynAIS.
- IN_LOOP: The event belongs to an already detected loop, but it is not the event identified as the “first” element in an iteration.
- NEW_ITERATION: The event is identified is the “first” element in a loop and it is not the first iteration
- END_LOOP: The previous event was an event in al already detected loop, but this one breaks to the known events in the sequence of the loop.
- END_NEW_LOOP: special case, where the event has been identified by DynAIS as an event that ends a previous loop and starts a new one at the same time.

2.3 EAR lifecycle management

ear_base implements the EAR lifecycle and it is used by before_mpi function. Table 2 shows the functions that controls the EAR lifecycle. The core functionality, the one that implements most of the EAR states and transitions, is the states_new_iteration.

Function	**Description**
void states_begin_job(...)	Executed at application start. Reads EAR configuration env. variables
void states_begin_period(...)	Executed at NEW_LOOP or END_NEW_LOOP DynAIS event
void states_new_iteration(...)	Executed at NEW_ITERATION DynAIS event.
void states_end_period(...)	Executed at END_LOOP or END_NEW_LOOP DynAIS event
void states_end_job	Executed at application end

Table 2: EAR is driven by DynAIS states

EAR implements a state diagram with the following states:

- NO_PERIOD

- FIRST_ITERATION
- EVALUATING_SIGNATURE
- SIGNATURE_STABLE
- RECOMPUTING_N
- SIGNATURE_HAS_CHANGED

Figure 6 shows transitions between states, the events that generate these transitions and main actions associated with changes. This diagram is implemented in the context of the EAR_NEW_ITERATION function. This function is called each time DynAIS reports a new iteration.

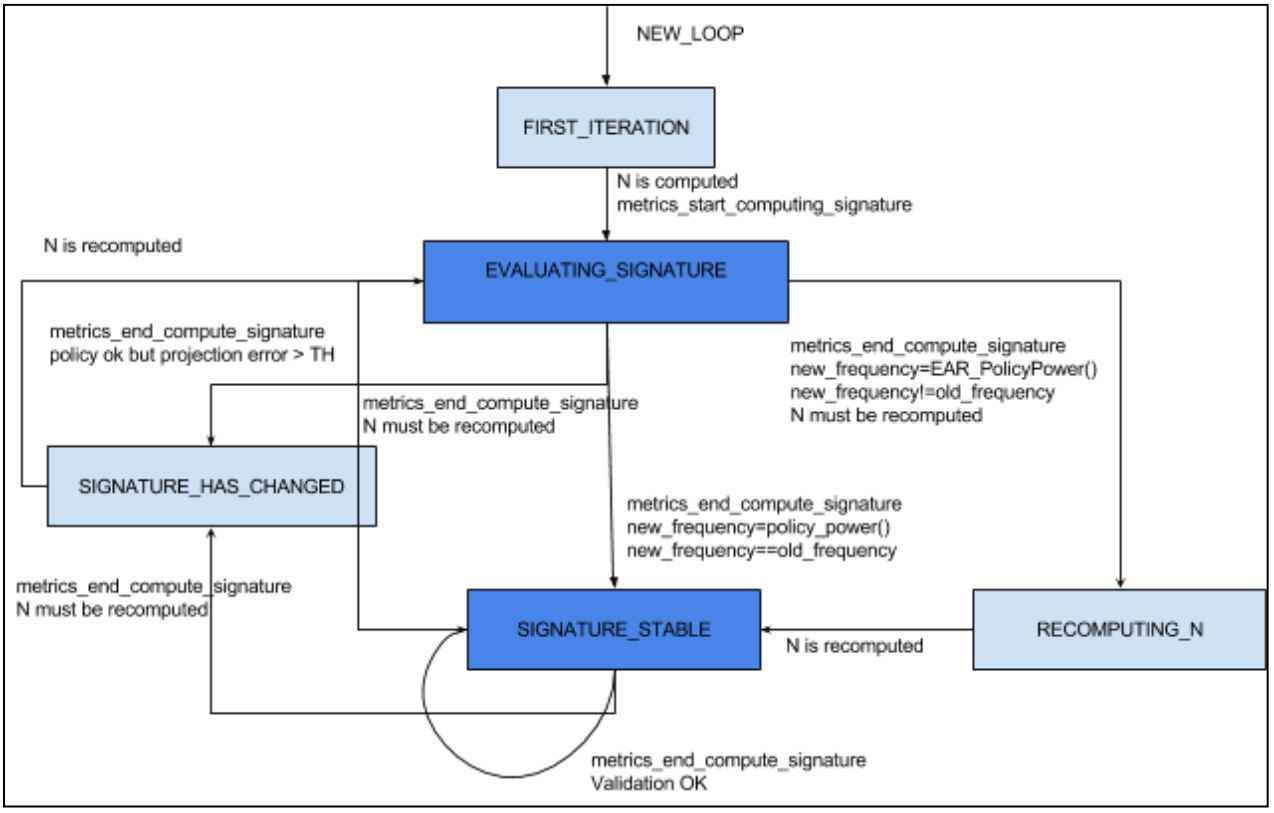

Figure 6: Main EAR states and transitions

States in dark blue are states where transitions from them are only allowed after N NEW_ITERATION events. States in light blue occurs after a NEW_ITERATION event. We removed these transitions just for clarity.

When a new is detected, we enter in the **FIRST_ITERATION** state. The state is in charge of computing N, the number of minimum iterations required to compute the application signature. This value is architecture dependent, since it depends basically on power meters capabilities. It is not automatically detected, it must be specified using the EAR_PERFORMANCE_ACCURACY environment variable.

After one NEW_ITERATION event, EAR starts computing the application signature (metrics_start_computing_signature) from *ear_metrics*, see section 2.4) and goes to the **EVALUATING_SIGNATURE** state. EAR will remain in this state for N iterations. After N iterations, the signature is ready to be computed (metrics_end_compute_signature from ear_metrics). This function stops and read performance counters while computing

Application signature metrics (Time, CPI, TPI and Power), see section 2.4 . After computing the application signature, next state will be one of these:

- SIGNATURE_HAS_CHANGED: The execution time of N iterations has been resulting not being enough to compute the application signature and N must be recomputed. This problem could happen when the first iteration has a bigger execution time than the rest. If that happens, next state will be SIGNATURE_HAS_CHANGED.
- SIGNATURE_STABLE: After computing the application signature, EAR executes the Energy Policy and the frequency selected is the current frequency. In that case, there is no need to recompute anything since we will continue at the same frequency.
- RECOMPUTING_N: After computing the application signature, EAR executes the Energy Policy and the frequency selected is not the current frequency. In that case, EAR will recompute N since the application will, potentially, run slower.

If EAR goes to **RECOMPUTING_N**, after one iteration N will be recomputed and EAR will go to SIGNATURE_STABLE state.

EAR will be in the **SIGNATURE_STABLE** state for at least N iterations, recomputing the application signature with the new frequency. After N iterations, EAR will compare new values for Time and power with the energy policy goals, validating the scheduling decisions.

If performance metrics are ok with the energy policy selected, and performance projections made by the policy are “similar” to the new performance metrics, we will continue in this state. In that case we double the number of iterations to be considered for the next validation. In that way, for a very regular application, we minimize the EAR overhead.

If EAR detects the energy policy decision was not correct or performance projections are not ok, next state will be EVALUATING_SIGNATURE since it is the state where the energy policy is executed. However, in some cases where EAR detects the application behavior has changed, next state will be SIGNATURE_HAS_CHANGED.

The **SIGNATURE_HAS_CHANGED** just recomputes N and goes to EVALUATING_SIGNATURE state.

2.4 EAR performance metrics

EAR is continuously monitoring the application. Performance and power metrics are collected at the beginning and end of loop iterations. EAR dynamically computes the number of iterations (for any detected loop) that must be considered in order to have representative performance values. The minimum time for performance accuracy is defined through the EAR_PERFORMANCE_ACCURACY environment variable. Table 3 shows metrics considered in the application signature. To compute TPI, CPI, and POWER EAR uses the *ear_daemon* API, since Total_Uncore_cas_count_RD and Total_Uncore_cas_count_WR counters and node dc energy needs root privileges.

Metric	**Description**
Time	Iteration time (in microseconds)

CPI	Average CPI (Cycles per instructions), It is computed as : CPI = Total_cycles / Total_instructions
TPI	Average TPI (Transactions per instruction). It is computed as: TPI = (Total_Uncore_cas_count_RD + Total_Uncore_cas_count_WR) / (Total_instructions / Cache_line_size)
GBS	Average GBS (GB/seconds). It is computed as: GBS = MEM_BW_READS + MEM_BW_WR MEM_BW_READS = (Total_Uncore_cas_count_RD * Cache_line_size) / (seconds*1024^3) MEM_BW_WR = (Total_Uncore_cas_count_WR * Cache_line_size) / (seconds*1024^3)
POWER	Average Power (Watts) (based on DC Node energy), It is computed as Total_energy / (Seconds*1000000). Energy was reported in uJ

Table 3: EAR metrics

- Total_cycles and Total_instructions are collected using PAPI counters ix86arch::UNHALTED_CORE_CYCLES and ix86arch::INSTRUCTION_RETIRED respectively [papi]
- Total_Uncore_cas_count_RD and Total_Uncore_cas_count_WR are collected reading uncore counters with an EAR code (see the EAR internals document) [uncore]

2.4.1 *ear_papi* API

ear_papi functions (table 4) are used by ear_base file implementing the EAR lifecycle. These functions abstract *ear_base* of low levels details about how to configure/start/stop/read performance counters.

Function (ear_papi.h)	Description
int metrics_init(...)	Initializes performance metrics: PAPI(cycles and instructions), and though ear_daemon (RAPL and Uncore counters for memory transactions) Returns 0 on success and 1 if id!=0 (not a "master"process)
void metrics_end(...)	Stops and release resources allocated to deal with performance metrics. Asks *ear_daemon* to stop counters. Prints summarized information.
long long metrics_time()	Returns time stamp. It is used to compute N.

void metrics_start_computing_signature(...)	Starts counters to compute application signature
struct App_info* metrics_end_compute_signature(...)	Stops and reads counters and computes application signature. Since application is supposed to be iterative, this function, after computing the Application Signature, starts counters again. It returns the application signature.

Table 4: EAR performance metrics API

metrics_start_computing_signature and metrics_end_computing_signature are the core functions exported by ear_metrics. Since it is supposed they are called in an iterative code, metrics_start_computing_signature is only executed when a new loop is detected.

2.4.2 ear_node_energy API

ear_node_energy provides the basic functions (table 5) to initialize node energy metrics, read accumulated dc node energy, compute the difference between two metrics and release resources allocated to node energy metrics.

Function	**Description**
void init_dc_energy()	Contacts ear_daemon to initialize node metrics
unsigned long read_dc_energy()	Returns the accumulated dc node energy Energy is reported in uJ.
unsigned long energy_diff(unsigned long EnerEnd, unsigned long EnerInit)	Returns the difference between two energy measurements
int end_dc_energy()	Contacts ear_daemon to release node energy resources

Table 5: DC energy management API

2.5 Performance/power models and Energy policies

2.5.1 Energy policies

EAR implements two energy policies plus one (MONITORING_ONLY), that can be used for monitoring purposes. Energy policies and its pseudocode that implements them are described in the Energy Aware Runtime reference document. The file that offers power models and policy managements is ear_models/ear_models.c

The MIN_ENERGY_TO_SOLUTION policy is guided by the EAR_PERFORMANCE_PENALTY environment variable. The MIN_TIME_TO_SOLUTION arguments are controlled by two constants defined at installation time:

PERFORMANCE_GAIN and EAR_MIN_P_STATE. PERFORMANCE_GAIN specifies a minimum performance efficiency gain and will avoid using high frequency values without a minimum performance gain controlled by the admin. The policy will use max(EAR_MIN_PERFORMANCE_EFFICIENCY_GAIN, PERFORMANCE_GAIN). EAR_MIN_P_STATE defined the default frequency to be used when MIN_TIME_TO_SOLUTION is used.

2.5.2 ear_models API

ear_models offers the following functions (see table 6). These functions, apart from initialization, are called when processing the EVALUATING_SIGNATURE and SIGNATURE_STATE states.

Function (ear_models.h)	**Description**
void init_power_policy()	Reads POLICY related environment variables that configures EAR power policies.
void init_power_models(..)	Reads coefficient information (hardware signature) to be used later on performance projections.
unsigned long policy_power(..)	Given the computed application signature, applies the selected power policy and returns the “optimal” frequency.
struct PerfProjection * performance_projection(unsigned long f)	Returns the performance projection for a given frequency
unsigned int policy_ok(..)	Given a performance projection, the actual application signature and the last computed application signature values, returns true if policy decision has been the correct ones.
unsigned int performance_projection_ok(..)	Given a performance projection and an application signature returns true if they are “equal” (with a maximum % of difference, defined by EAR_ACCEPTED_TH)

Table 6: Energy policies API

2.6 Output generated

The Energy Aware Runtime reference document describes the three types of information reported by EAR: log messages generated in the stderr during the execution (including a final summary), per-user and global database files with summarized metrics, and application trace files generated during the application execution. These traces are targeted to visualize the application behavior or metrics during the execution, but it can be also used for a

post-mortem analysis, since traces are very simple and can be adapted to any existing tool such as Paraver [paraver].

Verbosity level at stderr is controlled using a macro defined in ear_verbose.h file. This macro also prevents that processes MPI rank different than 0 to generates messages.

Per-user database file information is generated at application end at metrics_end(..) function. It uses the EAR_DB_USER_PATHNAME environment variable to create one file per node. Fields included in the file allows the aggregation of information using any spreadsheet tool since job_id, username, nodename and application name is generated. This file is a text file where fields where the character ";" is used as separator[1].

Global data base contains the same information than per-user database but it includes information for all the users in the system. This information is generated by the ear_daemon since database file is created with root permissions, preventing users to access information of other users. ear_lib requests the ear_daemon to write it and application end. Also for security, information is generated read only and in binary mode. To be able to load it in a spreadsheet tool, EAR includes a simply tool that translate binary information to a csv file.

2.6.1 EAR trace files

Trace file generation is offered by the ear_gui/ear_gui file (see table 7). Trace file generation is statically selected defining a constant at compile time (EAR_GUI). When defining this constant, a different version of EAR library is generated: libEARgui.so. To control the trace file size EAR includes a constant (MIN_FREQ_FOR_SAMPLING), defined at ear_gui/ear_gui.h. This constant limits the amount of information reported in the case of loops with very small iteration time. In this case, events will be generated only after MIN_FREQ_FOR_SAMPLING ms, reducing the trace file size.

Trace files are generated during the execution at different instrumentation points:

- Application start/end
- Application lifecycle (reported at states EVALUATING_SIGNATURE and SIGNATURE_STABLE) :
    - Application signature computed
    - Application performance projection when energy policy is applied
    - Frequency selected
- Loop management: New_period/End_period/New_iteration detected.

Function (ear_gui.h)	**Description**
void gui_init(..) void gui_end(...)	Executed at application start/end (processMPI.c)
void gui_new_signature(...) void gui_PP(..) void gui_frequency(...)	Executed when application signature is computed at EVALUATING_SIGNATURE and SIGANTURE_STABLE states (ear_base.c)

[1] csv is a standard format widely used by all the spreadsheet tools

void gui_new_period(..) void gui_end_period(..) void gui_new_n_iter(...)	Executed each time a new loop is detected, the loop ends, or a new iteration is reported.

Table 7: EAR dynamic tracing API

2.7 Environment variables

Since EAR library is dynamically loaded with applications, there is no option to configure it passing arguments. EAR library behaviour is configured through environment variables. The Energy Aware Runtime reference includes the complete list of EAR environment variables.

Environment variables are read by EAR at different files:

- ear_dyn_inst/processMPI.c reads EAR_VERBOSE, EAR_LEARNING_PHASE, SLURM_STEP_NUM_NODES, EAR_USER_DB_PATHNAME, and EAR_TMP
- ear_base/ear_base.c reads EAR_PERFORMANCE_ACCURACY
- ear_metrics/ear_papi.c reads EAR_APP_NAME
- ear_models/ear_models reads EAR_POWER_POLICY, EAR_PERFORMANCE_PENALTY, EAR_MIN_PERFORMANCE_EFFICIENCY_GAIN, EAR_RESET_FREQ, EAR_INSTALL_PATH, EAR_COEFF_DB_PATHNAME, EAR_P_STATE. Uses PERFORMANCE_GAIN and EAR_MIN_P_STATE defined at installation time.
- ear_db/ear_db reads EAR_DB_PATHNAME, EAR_INSTALL_PATH, SLURM_JOB_ID, LOGNAME, and EAR_APP_NAME

3 EAR daemon

EAR Daemon is the EAR component in charge of providing any kind of service that requires privileged capabilities. Current version is conceived as an external process executed with root privileges, launched before application starts and finished when application ends. The daemon must be manually launched once per node or managed by the EAR SLURM plugin. The daemon is fully distributed, neither exists shared information nor coordination between the different instances executed in different nodes. The scripts provided with the EAR framework makes easy the ear_daemon execution to final users.

3.1 Communication with ear_lib

EAR Daemon process contacts with client application through named pipes. Current design only supports a single ear_daemon per node and a single client per node. Two named pipes per "set of services" are created, one to receive client requests and another one to send values or ack's.

Ear_daemon blocks waiting for requests. When some data is ready it calls one of the three (currently) supported services (see figure 7):

- **Frequency services**: (1) changes node frequency (all CPUs at the same frequency); (2) sets the performance governor (not privileged operation) and (3) computes the

frequency when using turbo (ear_turbo.c and ear_frequency.c implements functions for this service.

- **Uncore counters services to compute memory bandwidth (GBS)**: allows start/reset/stop/read the integrated memory control (iMC) uncore counters to compute the memory bandwidth of a period of time. It is not implemented the support for reading any other uncore event. The access to uncore counters depends on the architecture and just a few are supported. **uncore_architectures** folder contains the uncore implementations for different architectures whereas **ear_uncores** offers a generic interface.
- **RAPL services**: allows to start/reset/stop/read PAPI events to read package and DRAM energy. Like the uncore counters, the set of events is statically defined. RAPL events needs root privileges depending on PAPI's version and CPU model. In some cases, they can be read without root privileges. ear_rapl_metrics file implements functions for this service.
- **System miscellaneous** services: The current EAR version includes a single service in this group, the report of the information in the system database.
- **Energy node metrics** services: accumulated energy is reported using IPMI raw commands. ear_daemon selects the specific command based on the system we are executing on.

After the initial configuration, ear_daemon is then basically a loop that iterates waiting for some requests coming from the running library in the same node.

```
while((numfds_ready=select(numfds_req,&rfds,NULL,NULL,NULL))>0){
        if (numfds_ready>0){
        for (i=0;i<ear_daemon_client_requests;i++){
                if (FD_ISSET(ear_fd_req[i],&rfds)){
                        switch (i){
                                case freq_req:
                                        ear_daemon_freq();
                                        break;
                                case uncore_req:
                                        ear_daemon_uncore(num_counters);
                                        break;
                                case rapl_req:
                                        ear_daemon_rapl();
                                        break;
                                case system_req:
                                        ear_daemon_system();
                                        break;
                                case node_energy_req:
                                        ear_daemon_node_energy();
                                        break;
                                default:
                                ear_exit("EAR_DAEMON: Error, request received not supported\n");
                        }// switch
                }       // IF FD_ISSET
        } //for
        }
}
```

Figure 7: EAR daemon main loop for service selection

3.2 Frequency services

The ear_frequency.c includes a set of functions to automatically detect the list of P_STATEs in the architecture, to change the frequency, etc (see figure 8). These features are implemented using the libcpupower.so library, available in Linux distributions. Not all the functions implemented in ear_frequency.c are exported to ear_lib since they don't need special privileges and are replicated in the library.

Function (ear_frequency.h)	**Description**	**ear_lib**
void ear_cpufreq_init();	Initializes information about the available frequencies and hardware architecture. Current CPU governor is saved.	no
void ear_cpufreq_end();	CPU governor is restored.	no
unsigned long ear_cpufreq_get(unsigned int cpuid);	Returns CPU frequency for cpuid.	no
unsigned long EAR_cpufreq_set(unsigned int cpuid,unsigned long newfreq);	Set newfreq frequency for cpuid. Returns old frequency on success and 0 on error.	no
unsigned long ear_cpufreq_set_node(unsigned long newfreq);	Set newfreq frequency for all cpus in current node.Returns old frequency on success and 0 on error	yes
unsigned long ear_get_nominal_frequency();	Returns nominal frequency of current node.	no
unsigned int ear_get_num_p_states();	Returns the total number of P_STATEs in current node.	no
unsigned long *ear_get_pstate_list();	Returns a pointer to the first elements of the array of frequencies. (from 0..(pstates-1)).	no
unsigned long ear_get_freq(unsigned int i);	Returns the frequency associated with a given P_STATE.	no
unsigned long ear_max_f();	Returns the maximum frequency in the current node.	no
unsigned long ear_min_f();	Returns the minimum frequency in the current node.	no
unsigned int ear_get_pstate(unsigned long f);	Returns the P_STATE associated with a given frequency.	no

void ear_begin_compute_turbo_freq();	Computing the effective frequency when turbo is on needs root privileges. This function starts counters to compute frequency.	yes
unsigned long ear_end_compute_turbo_freq();	Stops counters to compute effective frequency when turbo is on and returns the computed frequency.	yes
void ear_set_turbo();	Selects the “performance” governor.	yes

Figure 8: EAR frequency API

3.3 Uncore CAS counters

Integrated memory control (iMC) uncore counters in current Intel architectures is implemented using the PCI interface [intel-uncores]. Old versions were implemented using MSR registers. ear_daemon/ear_uncores.c file offers a generic interface that, based on the architecture, calls the specific function. CPU models where EAR has been tested are: Intel(R) Xeon(R) CPU E5-2697 v3(Model 63),Intel(R) Xeon(R) CPU E5-2697 v4 (Model 79), and Intel(R) Xeon(R) Gold 6148 CPU (Model 85).

The uncore interface defines the following struct to point to the machine specific function (see code4).

```
struct uncore_op
{
    int (*init) (int cpu_model);
    int (*count) ();
    int (*reset) ();
    int (*start) ();
    int (*stop) (unsigned long long *values);
    int (*read) (unsigned long long *values);
    int (*dispose) ();
} pmons;
```

Code4: EAR uncore operations data structure

Table 8 shows the generic uncore counters API. This API is targeted to read iMC uncore counters, it doesn’t support any other performance counter.

Function (ear_uncores.h)	**Description**

int init_uncores(int cpu_model);	Initializes specific hardware for uncore counters. Returns the specific pmons.init or -1 on error.
int count_uncores();	Return the number of performance monitor counters needed to compute memory bandwidth. (return pmons.count)
int reset_uncores();	Freezes and resets all performance monitor (PMON) uncore counters. returns pmons.reset
int start_uncores();	Unfreezes all PMON uncore counters. returns pmons.start
int stop_uncores(unsigned long long *values);	Freezes all PMON uncore counters and gets it's values. The array has to be greater or equal than the number of PMON uncore counters returned by count_uncores() function. The returned values are the read and write bandwith values in index [i] and [i+1] respectively.returns pmons.stop
int read_uncores(unsigned long long *values);	Reads the PMON uncore counters. The array has to be greater or equal than the number of PMON uncore counters returned by count_uncores() function. The returned values are the read and write bandwith values in index [i] and [i+1] respectively.returns pmons.read
int dispose_uncores();	Release resources allocated at init. pmons.dispose

Table 8: Generic API to access iMC uncore counters to compute memory bandwidth

We have implemented an architectural dependent file that reads iMC uncore counters based on PCI devices. This API is implemented in uncore_architectures/pci_uncores.c

Function (uncore_architectures/pci_uncores.h)	**Description**
int pci_init_uncores(int cpu_model);	Scans PCI buses and allocates file descriptors memory. Returns the number of performance monitor counters on success or -1 on error.
int pci_count_uncores();	Returns the number of performance monitor counters or -1 on error. init_uncore_reading() have to be called before to scan the buses.

int pci_reset_uncores();	Freezes and resets all performance monitor (PMON) uncore counters. Returns 0 on success or -1 on error.
int pci_start_uncores();	Unfreezes all PMON uncore counters returns 0 on success or -1 on error.
int pci_stop_uncores(unsigned long long *values);	Freezes all PMON uncore counters and gets it's values. The array has to be greater or equal than the number of PMON uncore counters returned by pci_init_uncores() or pci_count_uncores() function. The returned values are the read and write bandwidth values in index [i] and [i+1] respectively. Returns 0 on success or -1 on error.
int pci_read_uncores(unsigned long long *values);	Gets the PMON uncore counters values. Returns 0 on success and -1 on error.
int pci_dispose_uncores();	Closes file descriptors and frees memory. Returns 0 on success and -1 on error.

Table 9: Specific API for PCI based uncore counters

3.4 RAPL services

RAPL services are implemented in ear_daemon/ear_rapl_metrics.c file. It offers the following API (table 10).

Function	**Description**
int init_rapl_metrics();	Initializes the PAPI library if needed and the rapl::DRAM_ENERGY:PACKAGE0, rapl:::DRAM_ENERGY:PACKAGE1, rapl:::PACKAGE_ENERGY:PACKAGE0, rapl:::PACKAGE_ENERGY:PACKAGE1 events returns 0 on success and -1 on error
int reset_rapl_metrics();	Resets (zeroes) rapl events returns 0 on success and -1 on error
int start_rapl_metrics();	Starts rapl events returns 0 on success and -1 on error
int stop_rapl_metrics(unsigned long long *values);	Stops rapl events and copies events values in values array. returns 0 on success and -1 on error
void print_rapl_metrics();	Prints rapl metrics in the stderr

	returns 0 on success and -1 on error

Table 10: API for RAPL access

3.5 System services

ear_daemon offers a simple service to EAR to write application summary information in a root protected database. This service is included in ear_daemon for security reasons.

3.6 Energy node services

Ear_daemon reads accumulated DC node energy values using IPMI raw commands. The FreeIPMI library is used to implement the access to IPMI interface [FreeIPMI]. The ear_node_energy.c file implements a generic API (see table 11) to support different systems whereas ear_node_energy.h offers a generic API. When initializing the ear_daemon, the specific configuration is set, and subsequents calls will use the per-system function.

Function (ear_node_energy_metrics.h)	Description
int node_energy_init()	Detects cpu model and platform and initialize data structures and devices if required
int count_energy_data_length();	Returns the data size length in bytes returned by read_dc_energy
int read_dc_energy(unsigned long *energy);	Update energy with the accumulated node DC node energy in mJ
int node_energy_dispose();	Releases resources allocated in node_energy_init

Table 11: Generic API for node energy management

Figure 9 shows the node energy operations data structure. The implementation for the different systems can be found in node_energy_metrics folder.

```
struct node_energy_op
{
    int (*node_energy_init) ();
    int (*count_energy_data_length)();
    int (*read_dc_energy) (unsigned long *energy);
    int (*read_ac_energy) (unsigned long *energy);
    int (*node_energy_dispose) ();
} node_energy_ops;
```


Figure 9: Specific node energy API

4 EAR learning phase

The EAR distribution includes all the components to execute the learning phase and compute the per-node matrix coefficients that composes the hardware signature. Requirements to execute the learning phase are:

- A set of pre-selected kernels: The ear_learning_phase/kernels folder includes sources and Makefiles for all the kernel. Since all of them are part of more general benchmark suites, EAR distribution includes the original benchmark suites folders and Makefiles. During the first step of the learning phase, these kernels and compiled and configured for the specific architecture. During this step binaries are copied at $EAR_INSTALL_PATH/bin/kernels.
- **learning_phase_compile.sh / learning_phase_execute.sh** ($EAR_INSTALL_PATH/etc): Script file(s) (for non-slurm and slurm systems) to automatically compile, test and execute the kernels configuration (ear_learning_phase folder). These scripts must be updated to specify the number of sockets and cores per socket. They automatically compile the kernels for the architecture configuration specified, execute the kernels in a single node, and reports the execution time of each one. The “optimal” execution time for each kernel, in order to avoid a time consuming learning phase, is between 1 and 2 minutes.
    - The document EAR installation guide describes in detail how to update these kernels in order to adapt their execution time and where the different Makefiles are located to update CFLAGS, compiler version etc for each specific system.
- **learning_phase_helper.sh** ($EAR_INSTALL_PATH/etc): Script file that helps both compile and execute learning phase scripts, to maintain its code clean and minimizing the variables that the administrator have to edit.
- **compute_coefficients.c** ($EAR_INSTALL_PATH/bin): It computes the matrix coefficients once kernels have been executed. It reads the database file name, provided as argument, and computes and generates hardware signature in the coefficients file name specified by argument (the matrix of coefficients used by performance and power models to compute performance and power projections). This command uses the GNU Scientific Library [gsl] to apply a linear regression with performance metrics and compute coefficients.

5 EAR utils

EAR includes one additional command to help dealing with EAR internals included in ear_utils folder.

- show_metrics ($EAR_INSTALL_PATH/bin): This command reads the content of the database file generated by the ear_daemon and writes it to the stdout in text (csv) format, ready to be loaded in any spreadsheet tool. This command requires as argument the database file name.

6 EAR Slurm plugin

EAR SLURM plugin (ear_slurm_plugin.so in installation folder) allows you to load and unload dynamically both the EAR library and EAR daemon for a *SLURM* job, next to its proper configuration files.

This plugin is a type of *SLURM* plugin named *Spank* [spank], which allows you to alter your launching job adding new arguments to srun, modifying environment variables or executing and closing programs during the job execution time.

6.1 Job launching steps

When SLURM launches a job, which in turn launches one or more binaries, it does it through a series of steps. These steps have an associated a *Spank* function, and when compiling a plugin with those symbols, SLURM loads dynamically the plugin overlapping thus those function calls. These steps pass through different SLURM programs, for example *srun*, *slurmd*, and also in different contexts, like the local machine where you are calling the *srun* or *sbatch* program or remote, your cluster computing nodes. Depending on the program or the step you could have different permissions.

This a list of ordered steps which EAR SLURM plugin is involved to:

1) slurm_spank_local_user_init()
Called in local context after sun arguments has been processed. It is used to read EAR SLURM configuration files and converting its content in environment variables.

The same function is called after in remote context. It is used to alter the environment appending EAR library to LD_PRELOAD variable.

2) slurm_spank_task_init_privileged()
Called in remote context and one time per task, resulting in 20 calls if you have 20 tasks per node, for example. It is used to init the EAR daemon, which has enough privileges to grant access to all node resources required by the library. For example, switching the frequency, reading the uncore counters or in protected folders.

3) slurm_spank_exit()
Called in remote context just once, when all the tasks are finished. Used to close the EAR demon communication in case in case something wrong happened during job's execution, and performing some file cleaning if needed.

You can see these steps next to SLURM's job execution in the following image (figure 10):

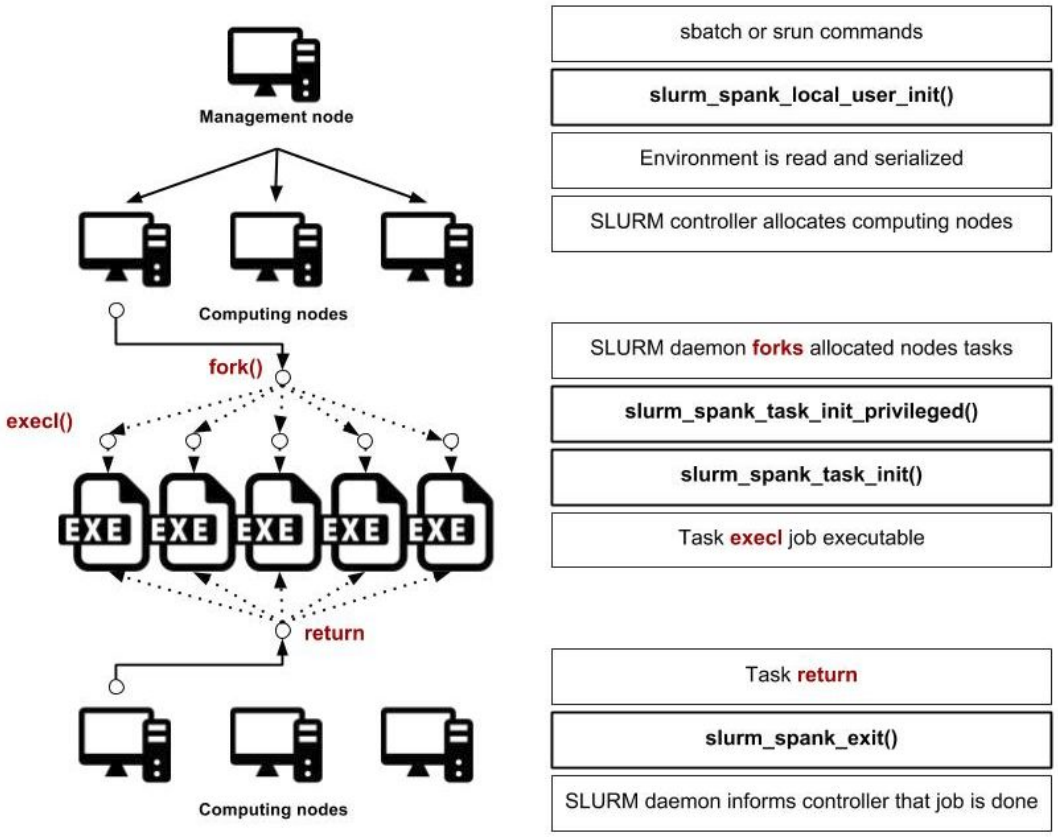

Figure 10: Main steps in EAR spank plugin for SLURM

6.2 Configuration file

As previously mentioned, a file, '*ear.conf*', is converted to environment variables which modifies the behavior of the library, daemon and plugin.

You can see an example of '*ear.conf*' in the figure 11:

```
EAR=0
EAR_DB_PATHNAME/etc/DBS/db
EAR_COEFF_DB_PATHNAME=/etc/COEFFICIENTS/coeff.
EAR_DYNAIS_LEVELS=4
EAR_DYNAIS_WINDOW_SIZE=200
EAR_LEARNING_PHASE=0
EAR_MIN_PERFORMANCE_EFFICIENCY_GAIN=0.75
EAR_PERFORMANCE_PENALTY=0.2
EAR_POWER_POLICY=MONITORING_ONLY
EAR_PERFORMANCE_ACCURACY=100000
EAR_RESET_FREQ=0
EAR_TURBO=0
EAR_TMP=/tmp
EAR_GUI_PATH=/tmp
EAR_VERBOSE=0
```

Figure 11. EAR configuration file for SLURM.

7 References

[slurm-spank] https://slurm.schedmd.com/spank.html.

[uncore] (Intel), R.D.: Documentation for uncore performance monitoring units, https://software.intel.com/en-us/blogs/2014/07/11/documentation-for-uncore-performance-monitoring-units.

[papi] http://icl.cs.utk.edu/papi/

[rapl] PAPITopics:RAPL Access - PAPIDocs, http://icl.cs.utk.edu/projects/papi/wiki/PAPITopics:RAPL_Access.

[naspb-mz] https://www.nas.nasa.gov/assets/pdf/techreports/2003/nas-03-010.pdf

[naspb-mpi] https://www.nas.nasa.gov/assets/pdf/techreports/1994/rnr-94-007.pdf

[naspb-ua] https://www.nas.nasa.gov/assets/pdf/techreports/2004/nas-04-006.pdf

[stream] http://www.streambench.org

[dgemm] https://software.intel.com/sites/default/files/mkl_fortran_samples_092017.tar_.gz

[bqcd] https://www.rrz.uni-hamburg.de/services/hpc/bqcd

[pmpi] http://mpi-forum.org/docs/mpi-3.1/mpi31-report/mpi31-report.htm#Node0

[intel-uncores] https://software.intel.com/en-us/blogs/2014/07/11/documentation-for-uncore-performance-monitoring-units

[paraver] https://tools.bsc.es/paraver

[gsl] https://www.gnu.org/software/gsl/

[FreeIPMI] https://www.gnu.org/software/freeipmi/

