Energy Aware Runtime (EAR): Reference guide

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

Energy Aware Runtime (EAR)

1 EAR GENERAL OVERVIEW

Energy Aware Runtime (EAR) aims also at finding the optimal frequency for a job according to energy policies (like Minimize Time to Solution or Minimize Energy to Solution) while being totally dynamic and transparent. EAR provides an “easy to use” and powerful solution to deal with the critical issue of energy management. Energy management is complicated since the is not a common platform, tool or mechanism for energy measurements and in many systems requires root privileges.

Before being able to use EAR with applications, we need to compute the System Signature during the installation phase. Once the system is installed or after some configuration change, EAR computes the System Signature. It’s a set of coefficients which characterize the performance and power consumption of each node in the system. System Signature is one of the inputs of Performance/power models to generate performance/power projections.[1]

This previous phase, prior to the normal EAR utilization, is called the learning phase, since it is a kind of system characterization. During the learning phase a matrix of coefficients per node is computed and stored in a set of files.

Once the Learning Phase is finished, we can execute EAR over MPI applications with the following main steps:

- First, EAR library (libEAR.so) relies on the profiling MPI mechanism and the LD_PRELOAD to transparently get control of MPI applications without having to modify or even recompile them. The mechanism allows EAR to be executed at any MPI call.
- Then, EAR uses a new technology to automatically detect the iterative structure existing in many HPC applications. This technology is called Dynamic Application Iterative Structure detection (DynAIS). DynAIS algorithm is invoked in each MPI call detecting repetitive sequences of events. EAR translates dynamic MPI calls (that is, considering MPI arguments) into DynAIS events, allowing DynAIS toidentify repetitive MPI calls with specific arguments during the application execution. That situation usually happens during loop execution.
- Once DynAIS has reported that a sequence is part of a repetition, also called loop, EAR gathers node metrics to compute performance and power for each loop iteration. After some iterations, perhaps one (potentially), EAR computes on the fly the Application Signature of that loop based on the following metrics: CPI, bandwith in GB/s, elapsed time and power.
- Based on the System Signature and Application Signature, EAR is able to predict the performance and power consumption for the next loop iterations for all the available frequencies in the node.

[1] See section 5 for more details on performance and power models

- Using these projections, and according to the Energy policy selected, EAR selects and applies the predicted optimal frequency for the next iterations.
- Finally, after some iterations have been executed at this optimal frequency, EAR recomputes the Application Signature. If some change is found, EAR recalculates the optimal frequency according to the new Application Signature.

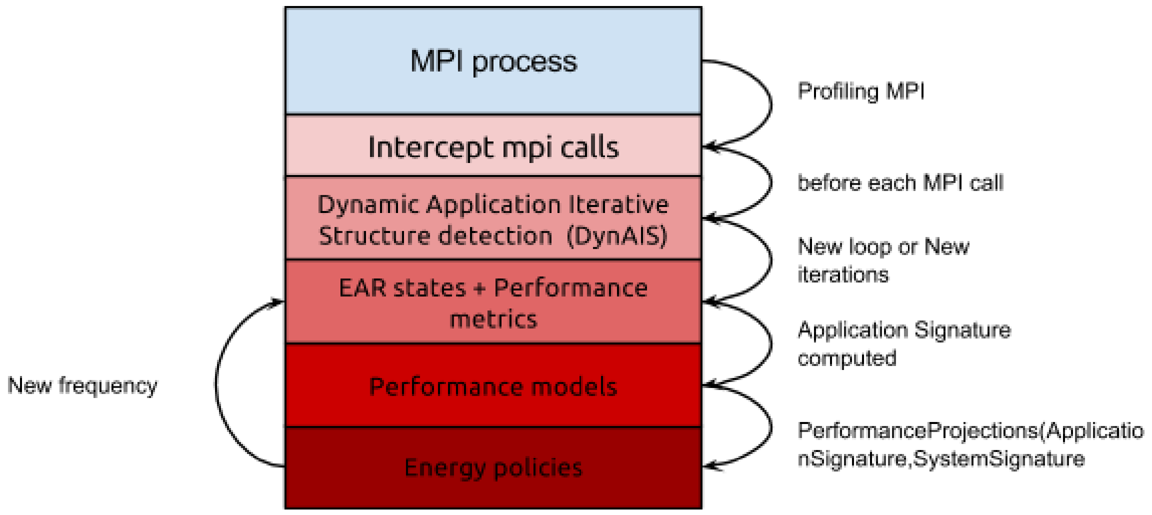

MPI process
Intercept mpi calls
Dynamic Application Iterative Structure detection (DynAIS)
EAR states + Performance metrics
Performance models
Energy policies
Profiling MPI
before each MPI call
New loop or New iterations
Application Signature computed
PerformanceProjections(ApplicationSignature,SystemSignature
New frequency

Figure 1 : EAR stack

Figure 1 summarizes the EAR stack for one process. More details about the different EAR steps are presented in the following sections.

Since EAR is a dynamically loaded library, there is not an API inserted in the application to invoke EAR. EAR configuration is set using environment variables (see section 8). EAR generates two types of outputs:

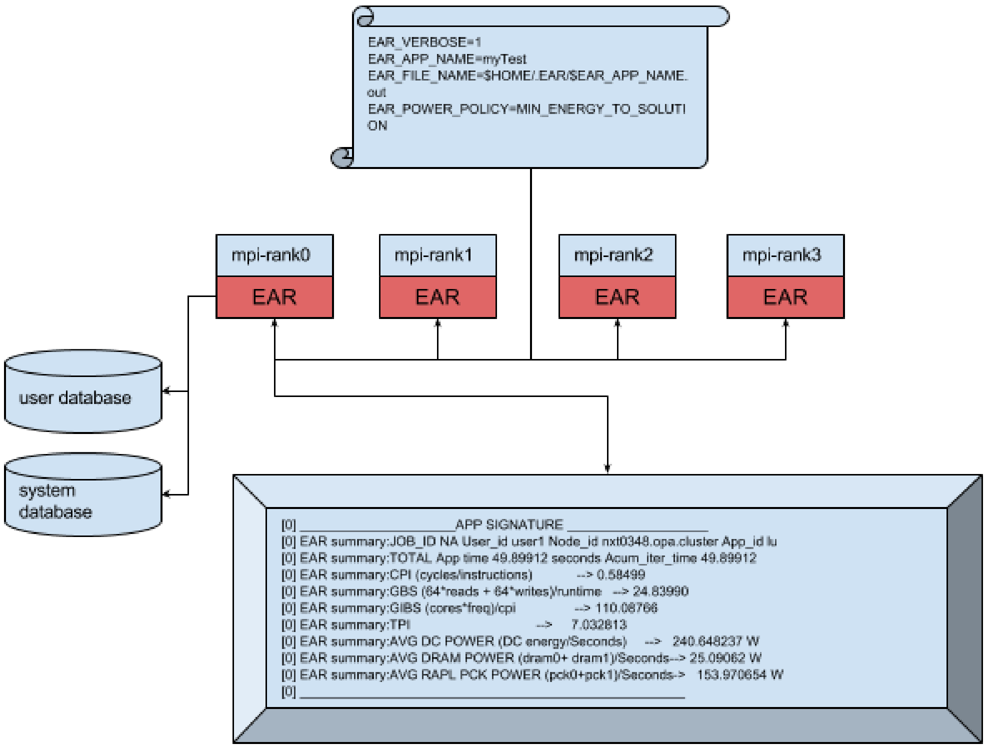
1. Messages generated during the execution showing the internal behavior of EAR (called verbose messages) and EAR traces.
2. Summary of the metrics generated at the end of the application execution.

First type of messages is also two fold:

1. Verbose messages: selected through the EAR_VERBOSE environment variable and reports information such as application and architecture basic information, loops detected, application signature computed, number of iterations executed, etc.
2. EAR traces: EAR includes a monitoring API that appends data in a set of files this reflecting application execution under EAR. These traces are designed to be used with the EAR GUI but they can be used with any other tool and easily converted to other formats such as Paraver [paraver]. More details can be found in the EAR internals document.

Second type, summarized metrics, are generated at application end. Application signature metrics are write at stderr together with a new line in a user specified csv file (filename is defined in EAR_USER_DB_PATHNAME environment variable). Moreover, a per-system database is updated at the end application end. Each column in the csv file contains the following values "USERNAME;JOB_ID;NODENAME;APPNAME;AVG.FREQ;TIME;CPI;TPI;GBS;DC-NODE-POWER;DRAM-POWER;PCK-POWER;DEF.FREQ;POLICY;POLICY_TH".

Figure 2 shows an overview of the the EAR input and outputs. More details can be found in section 8.



```
EAR_VERBOSE=1
EAR_APP_NAME=myTest
EAR_FILE_NAME=$HOME/.EAR/$EAR_APP_NAME.
out
EAR_POWER_POLICY=MIN_ENERGY_TO_SOLUTI
ON
```

```
[0] ______________________APP SIGNATURE ______________________
[0] EAR summary:JOB_ID NA User_id user1 Node_id nxt0348.opa.cluster App_id lu
[0] EAR summary:TOTAL App time 49.89912 seconds Acum_iter_time 49.89912
[0] EAR summary:CPI (cycles/instructions)           --> 0.58499
[0] EAR summary:GBS (64*reads + 64*writes)/runtime   --> 24.83990
[0] EAR summary:GIBS (cores*freq)/cpi               --> 110.08766
[0] EAR summary:TPI                           -->     7.032813
[0] EAR summary:AVG DC POWER (DC energy/Seconds)    -->   240.648237 W
[0] EAR summary:AVG DRAM POWER (dram0+ dram1)/Seconds--> 25.09062 W
[0] EAR summary:AVG RAPL PCK POWER (pck0+pck1)/Seconds->   153.970654 W
[0] ______________________________________________________________
```

Figure 2: EAR inputs and outputs

Following sections presents this different EAR steps:

- Intercepting the MPI calls
- Dynamic Application Iterative Structure detection
- EAR lifecycle management (EAR states) and performance metrics collection
- Performance/power models used for performance/power projections
- Energy policies

2 INTERCEPTING MPI CALLS

EAR library is dynamically and transparently loaded with MPI applications through the LD_PRELOAD mechanism using the standard profiling MPI interface (PMPI). MPI implements weak and strong symbols. Weak symbols can be replaced by user provided symbols, offering a simple way to intercept mpi calls, adding user code before and/or after strong symbols, or even replacing them. Each MPI call is wrapped by EAR and an input is generated to feed DynAIS algorithm. Figure 3 shows how each MPI call is intercepted by EAR when defining the LD_PRELOAD.

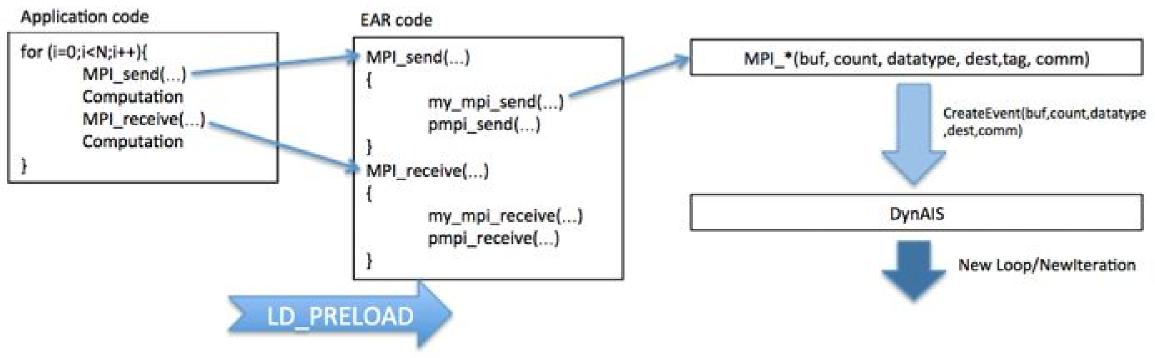


Figure 3 : Sequence of calls when using the PMPI interface to dynamically wrap MPI calls

Implementation details about the interception and event generation for DynAIS can be found in the Energy Aware Runtime internals document.

3 DYNAMIC APPLICATION ITERATIVE STRUCTURE DETECTION: DYNAIS

DynAIS is a generic algorithm that dynamically detects repetitive sequences of values. Also doesn't introduce any additional semantic to what these values are. DynAIS is used by EAR to detect repetitive sequences of dynamic MPI calls and its arguments. EAR creates an event identification by combining representative values for each MPI call: the call type, the addresses of buffers, the size of the message, etc.

These dynamically created events are the inputs of DynAIS. The algorithm, will check if an argument belongs or not to al already detected repetitive sequence of values (a loop).

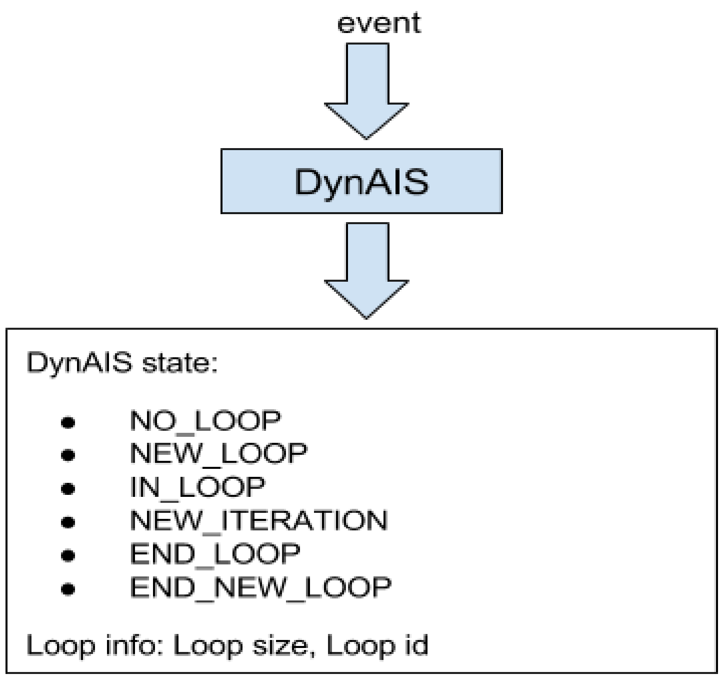


Figure 4: DynAIS states and returned values

DynAIS API also reports additional information such as the loop id and the loop size. DynAIS reported states will drive the EAR behavior. Since DynAIS is only providing the characterization of the application structure, any additional information such as iteration time or specific metrics must be collected by EAR. In particular, each time DynAIS returns BEGIN_NEW_LOOP or BEGIN_NEW_ITERATION, EAR executes its core functionality.

4 EAR STATES AND PERFORMANCE METRICS MEASUREMENTS

At the beginning of a new loop EAR calculates the number of iterations (N) compute the next Application Signature. The value of N depends on the iteration time and the minimum time for power and performance accuracy. Figure 5 shows the main EAR states and transitions between them: Computing the Application Signature, Scheduling, and Validation. There are some intermediate states not described here for simplic

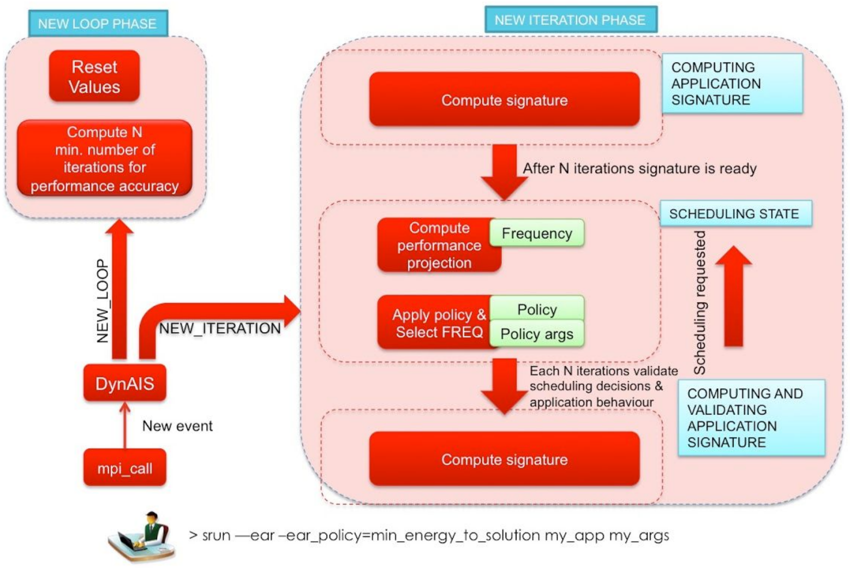

NEW LOOP PHASE
Reset Values
Compute N min. number of iterations for performance accuracy
NEW ITERATION PHASE
Compute signature
COMPUTING APPLICATION SIGNATURE
After N iterations signature is ready
SCHEDULING STATE
Compute performance projection
Frequency
Apply policy & Select FREQ
Policy
Policy args
Scheduling requested
NEW_LOOP
NEW_ITERATION
DynAIS
New event
mpi_call
Each N iterations validate scheduling decisions & application behaviour
COMPUTING AND VALIDATING APPLICATION SIGNATURE
Compute signature
> srun —ear –ear_policy=min_energy_to_solution my_app my_args

Figure 5: EAR lifecycle driven by DynAIS states reported

4.1 Computing Application Signature

Every N iterations EAR computes the Application Signature. The Application Signature includes: CPI, TPI, time and power (see section 7 for a detailed description of these metrics). Once DynAIS detects the main application structure, the Application Signature computed for one iteration of the outer loop will be representative for the whole application. This Application Signature is computed with the default frequency, or with the current frequency in case EAR had changed it in a previous loop. This is not a problem since EAR can project time and power using as reference frequency any frequency in the system. EAR uses this approach since most of times a new loop is, in fact, part of a bigger loop. In those cases, the Application Signature in the new loop reports very similar values than the old one, resulting in the same scheduling decisions.

Note: Computing performance metrics such as TPI and power could require privileged capabilities. To fix that problem, the EAR library uses one external process per node executed with root privileges.

4.2 Scheduling state

The first time EAR computes the Application Signature for a given loop, it computes the performance and power projections for all the available frequencies using Application Signature and System Signature. Given these projections and the Energy policy selected (see section 4.2 for more information about energy policies), EAR selects the "optimal" frequency for, at least, the next N iterations.

4.3 Computing and Validating Application Signature state

Once applied the Energy policy, EAR goes on computing Application Signature to detect (potential) changes in the same loop and miss predictions of performance projections that affect the energy policy goal. Also to detect those cases where local scheduling decisions are not taking benefit from the point of view of the whole application. If some of the situations are detected, EAR will go to the scheduling state, reapplying the energy policy and selecting a new (or not) "optimal" frequency. When EAR detects the application is reporting a stable Application Signature, it increases the number of iterations considered in the validation state, minimizing the runtime overhead.

5 PERFORMANCE MODELS

Performance/power models generates performance and power projections using as input Application and System Signature. Application Signature is computed by EAR on the fly, and System Signature is computed by EAR in the learning phase.

System signature is composed by six coefficients that characterize the architecture for a given frequency and a specific Reference frequency (Rf). Three of them (A(Rf,fn), B(Rf,fn) C(Rf,fn)) model the power projection and three of them (D(Rf,fn),E(Rf,fn),E(Rf,fn)) model CPI behavior (Time projections are done based on CPI projections).

$$Power(Rf,fn) = A(Rf,fn) * Power(Rf) + B(Rf,fn) * TPI(Rf) + C(Rf,fn)$$

$$CPI(Rf,fn) = D(Rf,fn) * CPI(Rf) + E(Rf) * TPI(Rf) + F(Rf,fn)$$

$$Time(Rf,fn) = Time(Rf) * CPI(Rf,fn)/CPI(Rf) * (Rf/fn)$$

The reference frequency (Rf) is the current CPU frequency and Fn is the frequency to be projected. With these models, we can compute performance and power projections independently of the frequency we are executing and the frequency to be projected. These models have shown to provide an average error of 5%.

6 ENERGY POLICIES

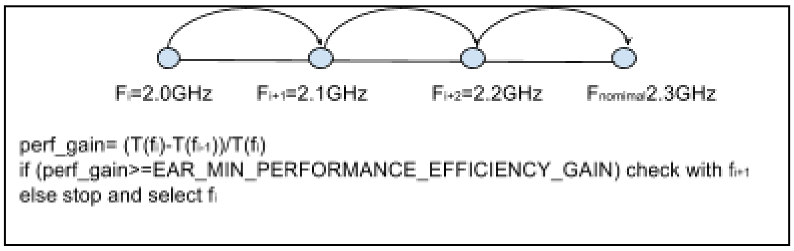
EAR implements two energy policies and a third option (NO_POLICY) that can be used for monitoring purposes:

- MIN_ENERGY_TO_SOLUTION: The goal is to minimize the energy consumed with a limiting performance degradation. The limit in the performance degradation is defined in the environment variable EAR_PERFORMANCE_PENALTY. Once N loop iterations are executed (where N is dynamically set as the number of iterations that fits the performance accuracy requirements), EAR computes the application signature. Based on the application and hardware signature, EAR computes a performance projection for each available frequency. This policy will select the optimal frequency that minimizes energy enforcing (performance_degradation <= EAR_PERFORMANCE_PENALTY). When executed with MIN_ENERGY_TO_SOLUTION policy, applications starts at nominal frequency.

```
bestFreq=defaultFreq
refTime=Time(defaultFreq)
Stop=0
// nexFreq goes from lower to higher frequencies
For (i=nextFreq(defaultFreq);(i<=MaxFreq && !stop) ;i=nextFreq(i)){
  newTime=ProjectTime(i)
  perf_gain=(refTime-newTime)/refTime;
  efficiency_gain=((i-bestFreq)/bestFreq)*EAR_PERFORMANCE_PENALTY
  if (perf_gain<efficiency_gain) stop=1
  else {
    refTime=newTime
    bestFreq=i
  }
}
```

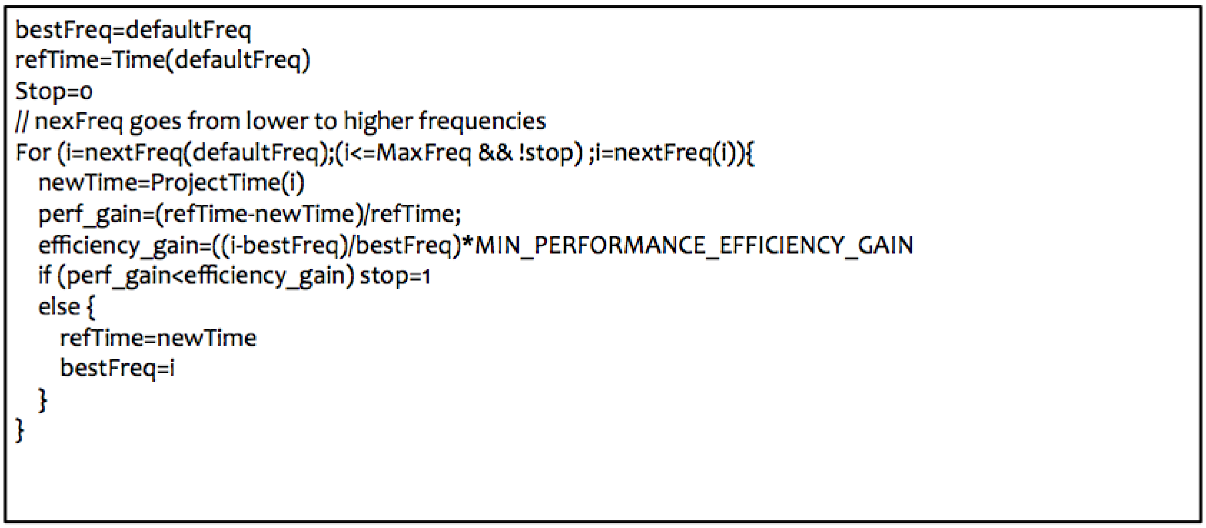
Figure 6: Minimize energy to solution algorithm

- MIN_TIME_TO_SOLUTION: The goal is to improve the execution time while guaranteeing a minimum performance efficiency that justifies more power consumption. The user can specify a minimum performance efficiency gain by setting the EAR_MIN_PERFORMANCE_EFFICIENCY_GAIN environment variable. This variable will prevent applications that do not scale with frequency, consuming more energy for nothing. For system efficiency, this gain must be greater or equal than a system defined value. Applications executed with this policy are started at lower frequency than nominal and EAR will will increase the frequency if performance_gain >= EAR_MIN_PERFORMANCE_EFFICIENCY_GAIN. When executed with MIN_TIME_TO_SOLUTION policy, applications starts at a predefined frequency lower than nominal (EAR_MIN_P_STATE defined at EAR installation time).For example, given a system with a nominal frequency of 2.3GHz and EAR_MIN_P_STATE set to 3, an application executed with MIN_TIME_TO_SOLUTION will start with frequency F_0=2.0Ghz (3 p_states less than nominal). When application metrics are computed, the library will compute performance projection for F_{i+1} and will compute the performance gain as shown in figure 7. If performance gain is greater or equal than EAR_MIN_PERFORMANCE_EFFICIENCY_GAIN, the policy will check with the next performance projection F_{i+2}. If the performance gain computed is less than EAR_MIN_PERFORMANCE_EFFICIENCY_GAIN, the policy will select the current frequency. This is shown on the diagram below with the example with a nominal frequency of 2.3 GHz and EAR_MIN_P_STATE set to 3. If the performance gain efficiency criteria is never satisfied, the frequency will stay at 2.0 GHz.



perf_gain= $(T(f_i)-T(f_{i+1}))/T(f_i)$
if (perf_gain>=EAR_MIN_PERFORMANCE_EFFICIENCY_GAIN) check with f_{i+1}
else stop and select f_i

Figure 7: MIN_TIME_TO_SOLUTION uses EAR_MIN_PERFORMANCE_EFFICIENCY_GAIN as the minimum value for the performance gain between between Fi and Fi+1



```
bestFreq=defaultFreq
refTime=Time(defaultFreq)
Stop=0
// nexFreq goes from lower to higher frequencies
For (i=nextFreq(defaultFreq);(i<=MaxFreq && !stop) ;i=nextFreq(i)){
  newTime=ProjectTime(i)
  perf_gain=(refTime-newTime)/refTime;
  efficiency_gain=((i-bestFreq)/bestFreq)*MIN_PERFORMANCE_EFFICIENCY_GAIN
  if (perf_gain<efficiency_gain) stop=1
  else {
    refTime=newTime
    bestFreq=i
  }
}
```

Figure 8: Minimize time to solution algorithm

- MONITORING_ONLY: when selecting MONITORING_ONLY, the selection of a new frequency is disabled but the rest of the ear library functionality is used (i.e. DynAIS).

7 EAR PERFORMANCE AND POWER METRICS

EAR is continuously monitoring the application. Performance and power metrics are collected at the beginning and at the 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. Table 1 summarizes main metrics computed by EAR. These metrics (except GBS) composes the application signature.

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 is reported in uJ

Table 1: Main metrics computed by EAR

8 EAR CONFIGURATION: ENVIRONMENT VARIABLES

Since EAR library is dynamically loaded with applications there is no option to configure it while passing it arguments. The library behaviour is configured then using environment variables. Table 2 shows complete list of environment variables EAR library uses. Some of them configure basic options such as the energy policy, which are supposed to be (potentially) modified by users from one to another application execution. The rest of variables allows advanced users (or EAR developers) to configure EAR behaviour to evaluate the performance impact of very specific features. EAR assumes that none of them could be user defined, so EAR uses default values for all of them.

Environment Variable	DESCRIPTION	Values
EAR_POWER_POLICY	Selects the energy policy. When NO_POLICY is selected, new frequency selection is disabled	(default)MIN_ENERGY_TO_SOLUTION, MIN_TIME_TO_SOLUTION, and MONITORING_ONLY
EAR_VERBOSE	Defines the verbose level. Higher the level, more messages. Messages are shown in stderr.	0..5 (int) default=0
EAR_MIN_PERFORMANCE_EFFICIENCY_GAIN	Minimum performance efficiency gain for MIN_TIME_TO_SOLUTION policy in the form of percentage (e.g. 0.75 meaning 75%). It prevents switching the CPU to the next value in the range of available frequencies, if the performance increase indicator do not scale over this efficiency gain value, preventing the waste of energy.	0..1 (float) Default=0.75
EAR_PERFORMANCE_PENALTY	Maximum performance degradation for MIN_ENERGY_TO_SOLUTION policy in the form of percentage	0..1 (float) Default=0.1

	(e.g. 0.15 meaning 15%). It prevents selection a frequency that is over this performance penalty.	
EAR_LEARNING_PHASE	Enables the learning phase→ Disables Dynais and energy policy is set to NO_POLICY	boolean. Default FALSE
EAR_APP_NAME	Application name used to save summarized metrics in DB	string. Default executable name obtained with papi
EAR_USER_DB_PATHNAME	File name with EAR specific summarized output. 1 line per application (1 file per node is generated)	filename path. Default $HOME/.ear_user_db.hostname
EAR_COEFF_DB_PATHNAME	Coefficients file name. (EAR library will concatenate the nodename and frequency)	database file path. Default $EAR_INSTALL_PATH/COEFFICIENTS/COEFFS_
EAR_DYNAIS_LEVELS	Number of levels of DynAIS algorithm (see section 3.2).	> 0 (int), default 7
EAR_DYNAIS_WINDOW_SIZE	Windows size of DynAIS algorithm (see section 3.2).	> 100 (int) default 200
EAR_PERFORMANCE_ACCURACY	Minimum time interval between two power measurements for a representative value.	> 0 (microseconds) default 1000000
EAR_RESET_FREQ	Set frequency to nominal at the beginning of a new loop or not.	boolean. Default FALSE
EAR_DB_PATHNAME	Defines a file name where summary metrics are stored at the application end. Ear lib creates one file per node. The nodename is concatenated to the EAR_DB_NAME value together with ".db" extension	Database file path. Default $EAR_INSTALL_PATH/DB/EAR_App_summary."*nodename*".db
EAR_TMP	Defines the path where temporary files are generated to connect libEAR.so and ear_daemon	path. Default $TMP
EAR_P_STATE	Defines the default frequency to be used when executing an application during the learning_phase or when MONITORING_ONLY policy is selected	valid p_state:1.. (max_p_states-1). Default 1
EAR_GUI_PATH	Defines the path where trace files are generated to connect libEARGUI.so and EAR_GUI tool	path.Default $EAR_TMP

Table 2: EAR environment variables

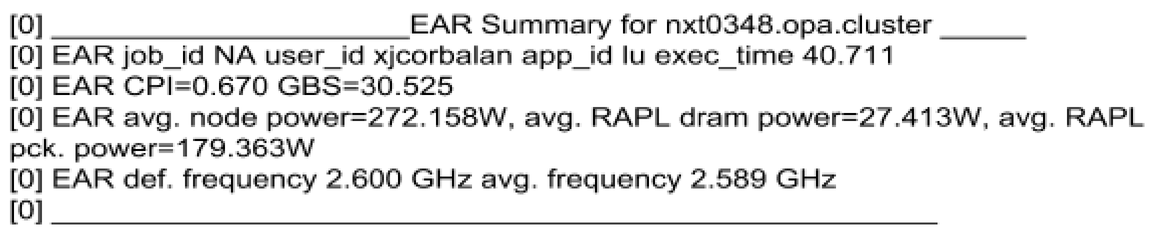
9 EAR OUTPUTS

9.1 Stderr messages

EAR users can select the level of verbosity by setting the EAR_VERBOSE environment value with the appropriate value. All verbose messages are generated in the stderr.

0) Only errors are reported.
1) Application granularity. EAR initial configuration is reported (policy, thresholds etc). Also basic information changes detected such as frequency or application signature.
2) Loop granularity is selected to report information (not per-iteration information).
3) Loop granularity with EAR lifecycle changes (EAR states) and metrics.
4) Iteration and function call granularity. Targets EAR debugging and validation purposes.

At application's end, independently of the verbose level selected, EAR reports (MPI rank 0) in the stderr a summary of metrics. Figure 9 shows an example of output.



```
[0] ______________________EAR Summary for nxt0348.opa.cluster ______
[0] EAR job_id NA user_id xjcorbalan app_id lu exec_time 40.711
[0] EAR CPI=0.670 GBS=30.525
[0] EAR avg. node power=272.158W, avg. RAPL dram power=27.413W, avg. RAPL
pck. power=179.363W
[0] EAR def. frequency 2.600 GHz avg. frequency 2.589 GHz
[0] _____________________________________________________
```

Figure 9: Information reported at application end

9.2 Per-user and global metrics DB

EAR also updates two files at application end with summarized information. One is a per-user database and other is a system-database. Per-user database is a csv text file with only jobs executed by current user. It can be used for benchmarking or performance analysis. It includes the following fields: USERNAME;JOB_ID;NODENAME;APPNAME; AVG.FREQ;TIME;CPI;TPI;GBS;DC-NODE-POWER;DRAM-POWER;PCK-POWER;DEF.FREQ;POLICY;POLICY_TH

- USERNAME;JOB_ID;NODENAME;APPNAME are reported for application identification.
  - Username as reported by LOGNAME environment variable
  - job_id as reported by SLURM_JOB_ID environment variable (NA otherwise)
  - nodename reported by gethostname system call
  - application name is either EAR_APP_NAME (if available) or executable name reported by papi.
- AVG.FREQ is average frequency computed by EAR
- TIME is application execution time. It will be the same in all the nodes
- CPI;TPI;GBS;DC-NODE-POWER;DRAM-POWER;PCK-POWER are average values for the current node
- DEF:FREQ is the default frequency used at submission time
- POLICY;POLICY_TH: In order to a better understanding of metrics, EAR reports the power policy at which application was executed and the policy threshold selected. For MONITORING_ONLY, policy threshold is set to 0.

Each node generates its own information in order to avoid overhead and synchronization. EAR creates one file per node with the name $EAR_FILE_NAME.nodename. Merging different files and using job_id as identifier, user can compute average power, total energy consumed, etc.

The system database includes the same fields but including information for all users in the system. EAR reports independently for security reasons. Moreover, information in this file is reported in binary mode for privacy reasons. System database attributes avoids user modifications or deletions. It is intended for system analysis. EAR creates one file per node with the name $EAR_DB_NAME.nodename.

9.3 EAR trace files

EAR library can generate trace files with detailed information about the application execution, in order to be post-processed or dynamically visualized with EAR graphic interface or any other analysis tool.In order to avoid overhead, this trace file is only generated when the EAR_GUI option is set at compile time, generating a new version of the library, libEARgui.so.

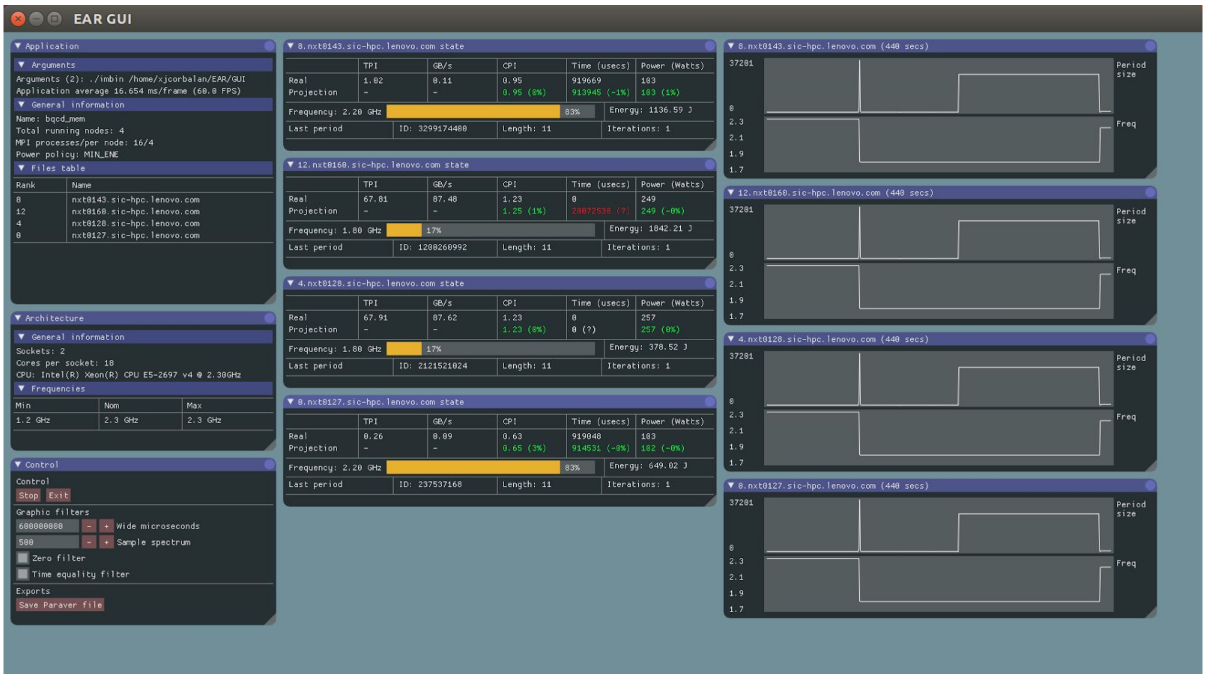
The EAR_GUI_PATH environment variable contains the path where trace files will be generated and is replaced by the user who doesn't want to place these files in a default location.

EAR generates two files describing the application and the architecture and one extra trace file per node. Application file includes basic information of the execution, such as application name, number of processes or energy policy selected. Architecture file includes information such as CPU model, number of sockets, cores per socket, etc.

Per-node Trace files are text file following a simple format: elapsed time; event; value. Events currently generated are the following ones:

- 1 = loop id
- 2 = loop size (number of MPI calls in loop)
- 3 = loop iteration number
- 4 = iteration time
- 5 = CPI
- 6 = TPI
- 7 = bandwidth GB/S
- 8 = power
- 9 = time projection
- 10 = CPI projection
- 11 = power projection
- 12 = current frequency
- 13 = total consumed energy

Figure 10 shows an example of the EAR GUI tool while visualizing the execution of a BQCD application [bqcd] executed in four nodes. The windows in the left side shows application and architecture information. EAR GUI also shows two windows per node: One is a summary showing current loop information and metrics and the other one is a timeline showing the loops sizes detected and the frequency selected over the time.


EAR GUI
8.nxt0143.sic-hpc.lenovo.com (440 secs)
12.nxt0160.sic-hpc.lenovo.com (440 secs)
4.nxt0128.sic-hpc.lenovo.com (440 secs)
0.nxt0127.sic-hpc.lenovo.com (440 secs)
37201
0
2.3
2.1
1.9
1.7
Period size
Freq

▼ Application

▼ Arguments

Arguments (2): ./imbin /home/xjcorbalan/EAR/GUI
Application average 16.654 ms/frame (60.0 FPS)

▼ General information

Name: bqcd_mem
Total running nodes: 4
MPI processes/per node: 16/4
Power policy: MIN_ENE

▼ Files table

Rank	Name
8	nxt0143.sic-hpc.lenovo.com
12	nxt0160.sic-hpc.lenovo.com
4	nxt0128.sic-hpc.lenovo.com
0	nxt0127.sic-hpc.lenovo.com

▼ Architecture

▼ General information

Sockets: 2
Cores per socket: 18
CPU: Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz

▼ Frequencies

Min	Nom	Max
1.2 GHz	2.3 GHz	2.3 GHz

▼ Control

Control
Stop Exit

Graphic filters
600000000 - + Wide microseconds
500 - + Sample spectrum
Zero filter
Time equality filter

Exports
Save Paraver file

▼ 8.nxt0143.sic-hpc.lenovo.com state

	TPI	GB/s	CPI	Time (usecs)	Power (Watts)
Real	1.02	0.11	0.95	919669	103
Projection	-	-	0.95 (0%)	913945 (-1%)	103 (1%)

Frequency: 2.20 GHz 83% Energy: 1136.59 J
Last period ID: 3299174400 Length: 11 Iterations: 1

▼ 12.nxt0160.sic-hpc.lenovo.com state

	TPI	GB/s	CPI	Time (usecs)	Power (Watts)
Real	67.81	87.48	1.23	0	249
Projection	-	-	1.25 (1%)	20072538 (?)	249 (-0%)

Frequency: 1.80 GHz 17% Energy: 1842.21 J
Last period ID: 1200260992 Length: 11 Iterations: 1

▼ 4.nxt0128.sic-hpc.lenovo.com state

	TPI	GB/s	CPI	Time (usecs)	Power (Watts)
Real	67.91	87.62	1.23	0	257
Projection	-	-	1.23 (0%)	0 (?)	257 (0%)

Frequency: 1.80 GHz 17% Energy: 378.52 J
Last period ID: 2121521024 Length: 11 Iterations: 1

▼ 0.nxt0127.sic-hpc.lenovo.com state

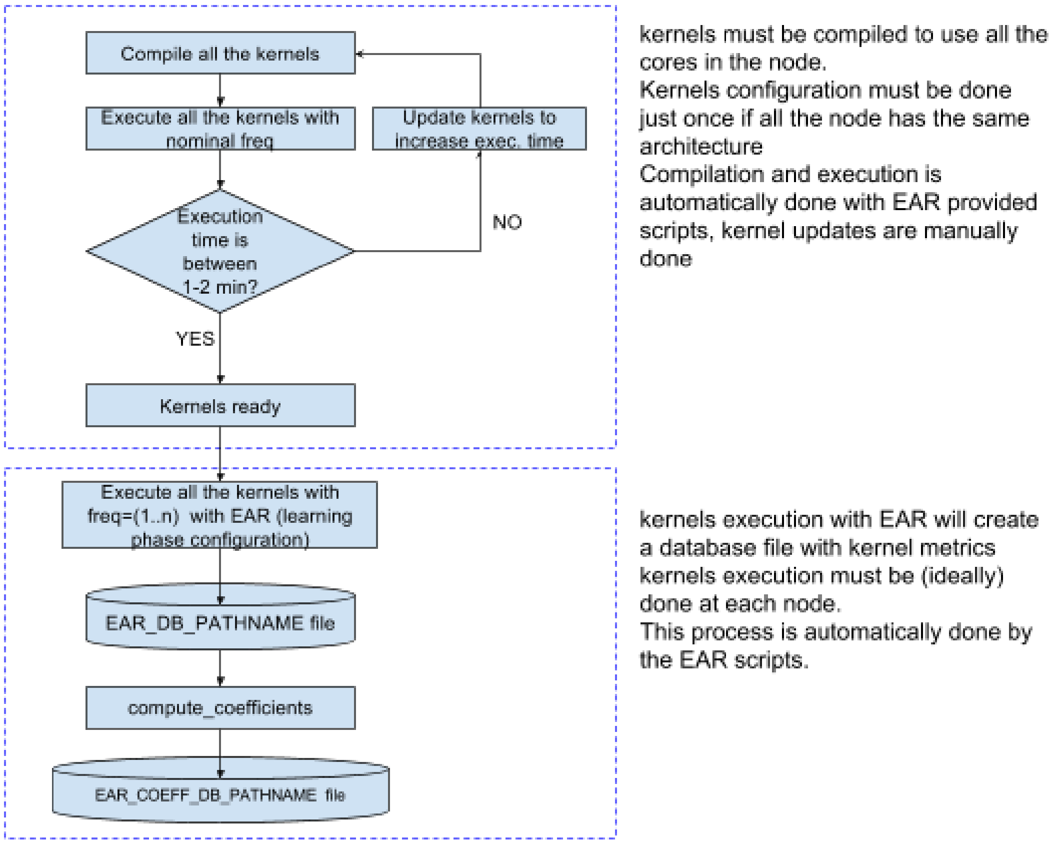
	TPI	GB/s	CPI	Time (usecs)	Power (Watts)
Real	0.26	0.09	0.63	919048	103
Projection	-	-	0.65 (3%)	914531 (-0%)	102 (-0%)

Frequency: 2.20 GHz 83% Energy: 649.02 J
Last period ID: 237537168 Length: 11 Iterations: 1

Figure 10: EAR Gui tool

10 EAR LEARNING PHASE: COMPUTING THE SYSTEM SIGNATURE

The learning phase executes a set of preselected kernels with a range frequencies of the system (potentially all of them), computes the application signature for each execution, and computes a matrix of coefficients characterizing the relationship between performance and power. These kernels must include applications with different characteristics, from memory bound to CPU bound applications. Figure 11 reflects main steps in the learning phase.


Compile all the kernels
Execute all the kernels with nominal freq
Update kernels to increase exec. time
Execution time is between 1-2 min?
NO
YES
Kernels ready
Execute all the kernels with freq=(1..n) with EAR (learning phase configuration)
EAR_DB_PATHNAME file
compute_coefficients
EAR_COEFF_DB_PATHNAME file

kernels must be compiled to use all the cores in the node.
Kernels configuration must be done just once if all the node has the same architecture
Compilation and execution is automatically done with EAR provided scripts, kernel updates are manually done

kernels execution with EAR will create a database file with kernel metrics
kernels execution must be (ideally) done at each node.
This process is automatically done by the EAR scripts.

Figure 11: EAR learning phase scheme

The list of selected kernels are:

- BT-MZ.C ,SP-MZ.C and LU-MZ.C from NPB3.3-MZ-MPI benchmark suite [naspb-mz]
- LU.C and EP.D from NPB3.3-MPI benchmark suite [naspb-mpi]
- UA.C from NPB3.3-OMP benchmark suite [naspb-ua]
- Stream [stream]
- dgemm [dgemm]

These benchmarks are configured to consume up to 2 min in modern architectures when using a single node. Details about how to update them to consume between 1 and 2 min per kernel can be found in the Energy Aware Runtime installation guide. [2]

Given that each kernel needs a maximum of 2 minutes of execution time, the learning phase for a range of 10 frequencies can be computed in 200 minutes as maximum. Typical values to compute a learning phase are less than 2 minutes per kernel and 6 different frequencies, resulting in less than 2 hours of learning phase since all the nodes can be done in parallel.

Ideally, the learning phase should be executed at each node since it could be small differences between nodes. However, EAR is prepared to use default coefficients when coefficients files of the current node are not available. That simplifies the process of executing the learning phase for sysadmins, not requiring the reservation of the whole system at a time.

[2] EAR distribution includes different scripts file to automatically compile the kernels and execute it for an easy tuning for each architecture

EAR distribution also includes a command to automatically submit the learning phase to N nodes, executing the kernels with the specified range of frequencies, and finally executing a command that computes a linear regression generating the matrix of coefficients, the system signature. The document EAR installation guide describes the steps to prepare and execute the learning phase when installing EAR in a system with scheduling support to EAR and without scheduling support. Current EAR version includes a plugin to be loaded with slurm workload manager.

11 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

[paraver] https://www.bsc.es/discover-bsc/organisation/scientific-structure/performance-tools