EAR

5.1

Generated by Doxygen 1.12.0

Chapter 1

Introduction

EAR 5.1 is a system software for energy management, accounting and optimization for super computers. Main EAR services are:

1. Application energy optimization. An **easy-to-use** and **lightweight** optimization service to automatically select the optimal CPU, memory and GPU frequency according to the application and the node characteristics. This service is offered by EAR core components: The EAR library and EAR Node Manager. EARL is a runtime library automatically loaded with the applications and it offers application metrics monitoring and it can select the frequencies based on the application behaviour on the fly. The Library is loaded automatically through the EAR Loader (EARLO) and it can be easly integrated with different system batch schedulers (e.g., SLURM). The EARL provides deep application accounting(both power/energy and performance) and energy optimization in a completelly transparent and dynamic way.
2. Job and Node monitoring: A complete **energy and performance accounting and monitoring system** . Node and application monitoring are also provided by the EAR core components (EAR library and EAR node manager). These two components are the data provides and information is reported to the DB using the EAR DB manager (EARDBD). The EARDBD is a distributed service offering buffering and aggregation of data, minimizing the number of connections with the DB server. EAR includes several report plugins for both relational (MariaDB and PostgreSQL) and non-relational Databases such as EXAMON. EAR commands for data reporting are only based on relational DBs.
3. Cluster power management (powercap). A **cluster energy manager** to monitor and control the energy consumed in the system through the EAR Global Manager (EARGMD) and EARD. EAR support a powerful and flexible configuration where different architectures with different power limits can be configured in the same cluster. CPU and CPU+GPU nodes are supported.

Visit the architecture page for a detailed description of each of these components. The user guide contains information about how to use EAR as an end user in a production environment. The admin guide has all the information related to the installation and setting up, as well as all core components details. Moreover, you can find a list of tutorials about how to use EAR.

1.1 License

EAR is an open source software licensed under EPL-2.0 license. Full text can be found in `COPYING.EPL-2.0` file distributed with the source code.

Contact: `ear-support@bsc.es`.

1.2 Publications

J. Corbalan, L. Alonso, J. Aneas and L. Brochard, "Energy Optimization and Analysis with EAR," 2020 IEEE International Conference on Cluster Computing (CLUSTER), 2020, pp. 464-472, doi: 10.1109/CLUSTER49012.2020.00067.

J. Corbalan, O. Vidal, L. Alonso and J. Aneas, "Explicit uncore frequency scaling for energy optimisation policies with EAR in Intel architectures," 2021 IEEE International Conference on Cluster Computing (CLUSTER), 2021, pp. 572-581, doi: 10.1109/Cluster48925.2021.00089.

J. Corbalan, L. Alonso, C. Navarrete and C. Guillen, "Soft Cluster Powercap at SuperMUC-NG with EAR," 2022 IEEE 13th International Green and Sustainable Computing Conference (IGSC), Pittsburgh, PA, USA, 2022, pp. 1-8, doi: 10.↩ 1109/IGSC55832.2022.9969360

Chapter 2

User guide

EAR was first designed to be usable 100% transparently by users, which means that you can run your applications enabling/disabling/tuning EAR with the less effort for changing your workflow, e.g., submission scripts. This is achieved by providing integrations (e.g., plug-ins, hooks) with system batch schedulers, which do all the effort to set-up EAR at job submission. By now, **`SLURM` is the batch scheduler full compatible with EAR** thanks to EAR's SLURM SPANK plug-in.

With EAR's SLURM plug-in, running an application with EAR is as easy as submitting a job with either `srun`, `sbatch` or `mpirun`. The EAR Library (EARL) is automatically loaded with some applications when EAR is enabled by default.

Check with the ear-info command if EARL is `on`/`off` by default. If it's `off`, use `--ear=on` option offered by EAR SLURM plug-in to enable it. For other schedulers, a simple prolog/epilog command can be created to provide transparent job submission with EAR and default configuration. The EAR development team had worked also with OAR and PBSPro batch schedulers, but currently there is not any official stable nor supported feature.

2.1 Use cases

Since EAR was targetting computational applications, some applications are automatically loaded and others are not, avoiding running EAR with, for example, bash processes. The following list resumes the application use cases where the EARL can be loaded transparently with them:

- MPI applications: IntelMPI, OpenMPI, Fujitsu and CRAY versions.
- Non-MPI applications: OpenMP, CUDA, MKL and OneAPI.
- Python applications.

Other use cases not listed here might be still supported. See the dedicated section.

2.1.1 MPI applications

EARL is automatically loaded with MPI applications when it is enabled by default (check `ear-info`). EAR supports the utilization of both `mpirun`/`mpiexec` and `srun` commands.

When using `sbatch`/`srun` or `salloc`, `Intel MPI` and `OpenMPI` are fully supported. When using specific MPI flavour commands to start applications (e.g., `mpirun`, `mpiexec.hydra`), there are some keypoints which you must take account. See next sections for examples and more details.

Review SLURM's `MPI Users Guide`, read your cluster documentation or ask your system administrator to see how SLURM is integrated with the MPI Library in your system.

2.1.1.1 Hybrid MPI + (OpenMP, CUDA, MKL) applications

EARL automatically supports this use case. `mpirun/mpiexec` and `srun` are supported in the same manner as explained above.

2.1.1.2 Python and Julia MPI applications

EARL cannot detect automatically MPI symbols when some of these languages is used. On that case, an environment variable is provided to give EARL a hint of the MPI flavour being used.

Export `EAR_LOAD_MPI_VERSION` environment with the value from the following table depending on the MPI implementation you are loading:

MPI flavour	Value
Intel MPI	*intel*
Open MPI	*open mpi* or *ompi*
MVAPICH	*mvapich*
Fujitsu MPI	*fujitsu mpi*
Cray MPICH	*cray mpich*

2.1.1.3 Running MPI applications on SLURM systems

Using srun command

Running MPI applications with EARL on SLURM systems using `srun` command is the most straightforward way to start using EAR. All jobs are monitored by EAR and the Library is loaded by default depending on the cluster configuration.

Even though it is automatic, there are few flags than can be selected at job submission. They are provided by EAR's SLURM SPANK plug-in. When using SLURM commands for job submission, both Intel and OpenMPI implementations are supported.

There is no need to load the EAR module for running a job with `srun` and get EARL loaded. Review SLURM's MPI Users Guide, read your cluster documentation or ask your system administrator to see how SLURM is integrated with the MPI Library in your system.

2.1.1.3.1 Using Using mpirun/mpiexec command

To provide an automatic loading of EARL, the only requirement from the MPI library is to be coordinated with the scheduler. Review SLURM's MPI Users Guide, read your cluster documentation or ask your system administrator to see how SLURM is integrated with the MPI Library in your system.

Intel MPI

Recent versions of Intel MPI offers two environment variables that can be used to guarantee the correct scheduler integrations:

- `I_MPI_HYDRA_BOOTSTRAP` sets the bootstrap server. It must be set to *slurm*.
- `I_MPI_HYDRA_BOOTSTRAP_EXEC_EXTRA_ARGS` sets additional arguments for the bootstrap server. These arguments are passed to SLURM, and they can be all the same as EAR's SPANK plug-in provides.

You can read here the Intel environment variables guide.

OpenMPI

For joining OpenMPI and EAR it is highly recommended to use SLURM's `srun` command. When using `mpirun`, as OpenMPI is not fully coordinated with the scheduler, EARL is not automatilly loaded on all nodes. Therefore EARL will be disabled and only basic energy metrics will be reported. To provide support for this workflow, EAR provides erun command. Read the corresponding examples section for more information about how to use this command.

2.1.1.3.1.1 MPI4PY

To use MPI with Python applications, the EAR Loader cannot automatically detect symbols to classify the application as Intel or OpenMPI. In order to specify it, the user has to define the `EAR_LOAD_MPI_VERSION` environment variable with the values specified in the table explained above.

It is recommended to add in Python modules to make it easy for final users. Ask your system administrator or check your cluster documentation.

MPI.jl

According to the `documentation`, the basic Julia wrapper for MPI is inspired by mpi4py. Check its section for running this kind of use case.

2.1.2 Non-MPI applications

2.1.2.1 Python

Since version 4.1 EAR automatically executes the Library with Python applications, so no action is needed. You must run the application with `srun` command to pass through the EAR's SLURM SPANK plug-in in order to enable/disable/tuning EAR. See EAR submission flags provided by EAR SLURM integration.

2.1.2.2 OpenMP, CUDA, Intel MKL and OneAPI

To load EARL automatically with non-MPI applications it is required to have it compiled with dynamic symbols and also it must be executed with `srun` command. For example, for CUDA applications the `--cudart=shared` option must be used at compile time. EARL is loaded for OpenMP, MKL and CUDA programming models when symbols are dynamically detected.

2.1.3 Other application types or frameworks

For other programming models or sequential apps not supported by default, EARL can be forced to be loaded by setting `EAR_LOADER_APPLICATION` enviroment variable, which must be defined with the executable name. For example:

```
#!/bin/bash

export EAR_LOADER_APPLICATION=my_app
srun my_app
```

2.1.4 Using EARL inside Singularity containers

Apptainer (formerly Singularity) is an open source technology for containerization. It is widely used in HPC contexts because the level of virtualization it offers enables the access to local services. It allows for geater reproducibility, making the programs less dependant on the environment they are being run on.

An example singularity command could look something like this:
```
singularity exec $IMAGE program
```

where IMAGE is an environment variable that contains the path of the Singularity container, and program is the executable to be run in the image.

In order to be able to use EAR inside the container two actions are needed:

- Binding EAR paths to make them visible in the container.
- Exporting some environment variables to the execution environment to make them available during the execution.

To bind folders there are two options: (1) using the environment variable SINGULARITY_BIND/APPTAINER_BIND or (2) using the -B flag when running the container. 1 is a comma separated string of pairs of paths [path_1][[:path_2][:perms]] such that *path_1* in local will be mapped into *path_2* in the image with the permissions set in *perms*, which can be *r* or *rw*. Specifying *path_2* and *perm* is optional. If they are not specified *path_1* will be bound in the same location.

To make EAR working the following paths sould be added to the binding configuration:

- $EAR_INSTALL_PATH,$EAR_INSTALL_PATH/bin,$EAR_INSTALL_PATH/lib,$EAR_TMP

You should have an EAR module to have the above environment variables. Contact with your system administrator for more information.

Once paths are deployed, to execute (for example) an OpenMPI application inside a Singularity/Apptainer enabling the EAR Library just the following is needed:
```
module load ear

mpirun -np <# processes> singularity exec $IMAGE erun --ear=on --program="program args"
```

A more complete example would look something like this:
```
export IMAGE=[path_to_image]/ubuntu_ompi.sif
export BENCH_PATH=[path_to_benchmark]
export APPTAINER_BIND="$EAR_INSTALL_PATH:$EAR_INSTALL_PATH:ro,$EAR_TMP:$EAR_TMP:rw"
export APPTAINERENV_EAR_REPORT_ADD=sysfs.so

mpirun -np 64 singularity exec $IMAGE $EAR_INSTALL_PATH/bin/erun \
    --ear=on --ear-verbose=1 \
    --program=$BENCH_PATH/bt-mz.D.64
```

Note that the example exports APPTAINERENV_EAR_REPORT_ADD to set the environment variable EAR_REPORT_ADD to load sysfs report plug-in. See next section about report plug-ins.

2.1.5 Using EARL through the COMPSs Framework

COMP Superscalar (COMPSs) is a task-based programming model which aims to ease the development of applications for distributed infrastructures, such as large High-Performance clusters (HPC), clouds and container managed clusters. COMPSs provides a programming interface for the development of the applications and a runtime system that exploits the inherent parallelism of applications at execution time. **Since version 5.0 EAR supports monitoring and optimization of workflows** and the COMPSs Framework includes the integration with EAR. Check out the dedicated section from the official COMPSs documentation for more information about how to measure the energy consumption of your workflows.

EARL loading is **only available** using enqueue_compss and with Python applications. The command has the flag --ear which you can set either a boolean (i.e., *true* or *false*) or a string value. The latter can be any of the [job submission flags](ear-job-submission-flags). See the example provided by the COMPSs documentation.

2.2 Retrieving EAR data

As a job accounting and monitoring tool, EARL collects some metrics that you can get to see or know your applications workload. The Library is doted with several modules and options to be able to provide different kind of information.

As a very simple hint of your application workload, you can enable EARL verbosity (i.e., `--ear-verbose=1`) to get loop data at runtime. **The information is shown at *stderr* by default.** Read how to set up verbosity at [submission time](ear-job-submission-flags) and `verbosity environment variables` provided for a more advanced tunning of this EAR feature.

2.2.1 Post-mortem application data

To get offline job data EAR provides **`eacct`** command, a tool to provide the monitored job data stored in the Database. You can request information in different ways, so you can read aggregated job data, per-node or per-loop information among other things. See `eacct usage examples` for a better overview of what `eacct` provides.

2.2.2 Runtime report plug-ins

There is another way to get runtime and aggregated data during runtime without the need of calling `eacct` after the job completion. EAR implements a reporting system mechanism which let developers to add new report plug-ins, so there is an unlimited set of ways to report EAR collected data. EAR releases come with a fully supported report plug-in (i.e., *csv_ts.so*) which provides the same runtime and aggregated data reported to the Database in CSV files, directly while the job is running. You can load this plug-in in two ways:

1. By setting `--ear-user-db` flag at submission time.
2. `Loading directly the report plug-in` through an environment variable: `export EAR_REPORT_ADD=csv_ts.so`.

Read [report plug-ins](Report) dedicated section for more information.

2.2.3 Other EARL events

You can also request EAR to report **events** to the [Database](EAR-Databse). They show more details about EARL internal state and can be provided with `eacct` command. See how to enable `EAR events reporting` and which kind of events EAR is reporting.

2.2.4 MPI stats

If your application applies, you can request EAR to report at the end of the execution a `summary about its MPI behaviour`. The information is provided along two files and is the aggregated data of each process of the application.

2.2.5 Paraver traces

Finally, EARL can provide runtime data in the `Paraver` trace format. Paraver is a flexible performance analysis tool maintained by the *Barcelona Supercomputing Center*'s tools team. This tool provides an easy way to visualize runtime data, computing derived metrics and to provide histograms for better of your application behaviour. See on the `environment variables page` how to generate Paraver traces.

Another way to see runtime information with Paraver is to use the open source tool **`ear-job-visualization`**, a CLI program written in Python which gets CSV files generated by `--ear-user-db` flag and converts its data to the Paraver trace format. EAR metrics are reported as trace events. Node information is stored as Paraver task information. Node GPU data is stored as Paraver thread information

2.3 EAR job submission flags

The following EAR options can be specified when running `srun` and/or `sbatch`, and are supported with `srun/sbatch/salloc`:

Option	Description
--ear=[on|off]	Enables/disables EAR library loading with this job.
--ear-user-db=<filename>	Asks the EAR Library to generate a set of CSV files with EARL metrics.
--ear-verbose=[0|1]	Specifies the level of verbosity; the default is 0.

When using `--ear-user-db` flag, one file per node is generated with the average node metrics (node signature) and one file with multiple lines per node is generated with runtime collected metrics (loops node signatures). Read eacct's section in the commands page to know which metrics are reported, as data generated by this flag is the same as the reported (and retrieved later by the command) to the Database.

Verbose messages are placed by default at *stderr*. For jobs with multiple nodes, `ear-verbose` option can result in lots of messages mixed at *stderr*. We recommend to split up SLURM's output (or error) file per-node. You can read SLURM's `filename pattern specification` for more information.

If you still need to have job output and EAR output separated, you can set `EARL_VERBOSE_PATH` environment variable and one file per node will be generated only with EAR output. The environemnt variable must be set with the path (a directory) where you want the output files to be generated, it will be automatically created if needed.

> You can always check the avaiable EAR submission flags provided by EAR's SLURM SPANK plug-in by typing `srun --help`.

2.3.1 CPU frequency selection

The EAR configuration file supports the specification of *EAR authorized users*, who can ask for a more privileged submission options. The most relevant ones are the possibility to ask for a specific optimisation policy and a specific CPU frequency.

> Contact with the sys admin or helpdesk team to become an authorized user.

- The `--ear-policy=policy_name` flag asks for *policy_name* policy. Type `srun --help` to see policies currently installed in your system.
- The `--ear-cpufreq=value` (*value* must be given in kHz) asks for a specific CPU frequency.

2.3.2 GPU frequency selection

EAR **version 3.4 and upwards** supports GPU monitoring for NVIDIA devices from the point of view of the application and node monitoring. GPU frequency optimization is not supported yet. **Authorized** users can ask for a specific GPU frequency by setting the SLURM_EAR_GPU_DEF_FREQ environment variable, giving the desired GPU frequency expressed in kHz. Only one frequency for all GPUs is now supported.

> Contact with sys admin or helpdesk team to become an authorized user.

To see the list of available frequencies of the GPU you will work on, you can type the following command:
```
nvidia-smi -q -d SUPPORTED_CLOCKS
```

2.4 Examples

2.4.1 srun examples

Having an MPI application asking for one node and 24 tasks, the following is a simple case of job submission. If EARL is turned on by default, no extra options are needed to load it. To check if it is on by default, load the EAR module and execute the ear-info command. EAR verbose is set to 0 by default, i.e., no EAR messages.
```
srun -J test -N 1 -n 24 --tasks-per-node=24 application
```

The following executes the application showing EAR messages, including EAR configuration and node signature in *stderr*.
```
srun --ear-verbose=1 -J test -N 1 -n 24 --tasks-per-node=24 application
```

EARL verbose messages are generated in the standard error. For jobs using more than 2 or 3 nodes messages can be overwritten. If the user wants to have EARL messages in a file the SLURM_EARL_VERBOSE_PATH environment variable must be set with a folder name. One file per node will be generated with EARL messages.
```
export SLURM_EARL_VERBOSE_PATH=logs
srun --ear-verbose=1 -J test -N 1 -n 24 --tasks-per-node=24 application
```

The following asks for EARL metrics to be stored in csv file after the application execution. Two files per node will be generated: one with the average/global signature and another with loop signatures. The format of output files is <filename>.<nodename>.time.csv for the global signature and <filename>.<nodename>.time.loops.csv for loop signatures.
```
srun -J test -N 1 -n 24 --tasks-per-node=24 --ear-user-db=filename application
```

For EAR *authorized users*, the following executes the application with a CPU frequency of 2.0GHz:
```
srun --ear-cpufreq=2000000 --ear-policy=monitoring --ear-verbose=1 -J test -N 1 -n 24 --tasks-per-node=24
      application
```

For --ear-cpufreq to have any effect, you must specify the --ear-policy option even if you want to run your application with the default policy.

2.4.2 sbatch + EARL + srun

When using `sbatch` EAR options can be specified in the same way. If more than one `srun` is included in the job submission, EAR options can be inherited from `sbatch` to the different `srun` instances or they can be specifically modified on each individual `srun`.

The following example will execute twice the application. Both instances will have the verbosity set to 1. As the job is asking for 10 nodes, we have set the `SLURM_EARL_VERBOSE_PATH` environment variable set to the *ear_log* folder. Moreover, the second step will create a set of csv files placed in the *ear_metrics* folder. The nodename, Job Id and Step Id are part of the filename for a better identification.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -e test.%j.err
#SBATCH -o test.%j.out
#SBTACH --ntasks=24
#SBATCH --tasks-per-node=24
#SBATCH --cpus-per-task=1
#SBATCH --ear-verbose=1

export SLURM_EARL_VERBOSE_PATH=ear_logs

srun application

mkdir ear_metrics
srun --ear-user-db=ear_metrics/app_metrics application
```

2.4.3 EARL + mpirun

2.4.3.1 Intel MPI

When running EAR with `mpirun` rather than `srun`, we have to specify the utilization of `srun` as bootstrap. Version 2019 and newer offers two environment variables for bootstrap server specification and arguments.

```
export I_MPI_HYDRA_BOOTSTRAP=slurm
export I_MPI_HYDRA_BOOTSTRAP_EXEC_EXTRA_ARGS="--ear-policy=monitoring --ear-verbose=1"
mpiexec.hydra -n 10 application
```

2.4.3.2 OpenMPI

Bootstrap is an Intel(R) MPI option but not an OpenMPI option. For OpenMPI `srun` must be used for an automatic EAR support. In case OpenMPI with `mpirun` is needed, EAR offers the `erun` command, which is a program that simulates all the SLURM and EAR SLURM Plug-in pipeline. You can launch `erun` with the `--program` option to specify the application name and arguments.

```
mpirun -n 4 /path/to/erun --program="hostname --alias"
```

In this example, `mpirun` would run 4 erun processes. Then, `erun` will launch the application `hostname` with its alias parameter. You can use as many parameters as you want but the semicolons have to cover all of them in case there are more than just the program name.

`erun` will simulate on the remote node both the local and remote pipelines for all created processes. It has an internal system to avoid repeating functions that are executed just one time per job or node, like SLURM does with its plugins.

IMPORTANT NOTE If you are going to launch `n` applications with `erun` command through a sbatch job, you must set the environment variable `SLURM_STEP_ID` to values from `0` to `n-1` before each `mpirun` call. By this way `erun` will inform the EARD the correct step ID to be stored then to the Database.

2.5 EAR job Accounting (eacct)

The eacct command shows accounting information stored in the EAR DB for jobs (and steps) IDs. The command uses EAR's configuration file to determine if the user running it is privileged or not, as **non-privileged users can only access their information**. It provides the following options.

2.5.1 Usage examples

The basic usage of `eacct` retrieves the last 20 applications (by default) of the user executing it. If a user is **privileged**, they may see all users applications. The default behaviour shows data from each job-step, aggregating the values from each node in said job-step. If using SLURM as a job manager, a *sb* (sbatch) job-step is created with the data from the entire execution. A specific job may be specified with `-j` option.

```
[user@host EAR]$ eacct -j 175966
    JOB-STEP USER       APPLICATION      POLICY NODES AVG/DEF/IMC(GHz) TIME(s)    POWER(W) GBS     CPI
       ENERGY(J)    GFLOPS/W IO(MBs) MPI%  G-POW (T/U)   G-FREQ  G-UTIL(G/MEM)
 175966-sb   user       afid             NP     2     2.97/3.00/---    3660.00    381.51   ---     ---
      2792619      ---      ---     ---   ---            ---     ---
 175966-2    user       afid             MO     2     2.97/3.00/2.39   1205.26    413.02   146.21  1.04
      995590       0.1164   0.0     21.0  ---            ---     ---
 175966-1    user       afid             MT     2     2.62/2.60/2.37   1234.41    369.90   142.63  1.02
      913221       0.1265   0.0     19.7  ---            ---     ---
 175966-0    user       afid             ME     2     2.71/3.00/2.19   1203.33    364.60   146.23  1.07
      877479       0.1310   0.0     17.9  ---            ---     ---
```

The command shows a pre-selected set of columns, read `eacct`'s section on the EAR commands page.

For node-specific information, the `-l` (i.e., long) option provides detailed accounting of each individual node↩ : In addition, `eacct` shows an additional column: `VPI(%)` (See the example below). The VPI is meaning the percentage of AVX512 instructions over the total number of instructions.

```
[user@host EAR]$ eacct -j 175966 -l
     JOB-STEP    NODE ID    USER ID    APPLICATION      AVG-F/IMC-F TIME(s)    POWER(s)   GBS        CPI
       ENERGY(J)  IO(MBS) MPI%  VPI(%)  G-POW(T/U)    G-FREQ G-UTIL(G/M)
  175966-sb       cmp2506    user       afid             2.97/---   3660.00    388.79     ---        ---
      1422970     ---      ---   ---     ---     ---      ---
  175966-sb       cmp2507    user       afid             2.97/---   3660.00    374.22     ---        ---
      1369649     ---      ---   ---     ---     ---      ---
  175966-2        cmp2506    user       afid             2.97/2.39  1205.27    423.81     146.06     1.03
      510807      0.0      21.2  0.23    ---     ---      ---
  175966-2        cmp2507    user       afid             2.97/2.39  1205.26    402.22     146.35     1.05
      484783      0.0      20.7  0.01    ---     ---      ---
  175966-1        cmp2506    user       afid             2.58/2.38  1234.46    374.14     142.51     1.02
      461859      0.0      19.4  0.00    ---     ---      ---
  175966-1        cmp2507    user       afid             2.67/2.37  1234.35    365.67     142.75     1.03
      451362      0.0      20.0  0.01    ---     ---      ---
  175966-0        cmp2506    user       afid             2.71/2.19  1203.32    371.76     146.25     1.08
      447351      0.0      17.9  0.01    ---     ---      ---
  175966-0        cmp2507    user       afid             2.71/2.19  1203.35    357.44     146.21     1.05
      430128      0.0      17.9  0.01    ---     ---      ---
```

If EARL was loaded during an application execution, runtime data (i.e., EAR loops) may be retrieved by using `-r` flag. You can still filter the output by Job (and Step) ID.

Finally, to easily transfer `eacct`'s output, `-c` option saves the requested data in CSV format. Both aggregated and detailed accountings are available, as well as filtering. When using along with `-l` or `-r` options, all metrics stored in the EAR Database are given. Please, read the commands section page to see which of them are available.

```
[user@host EAR]$ eacct -j 175966.1 -r
    JOB-STEP      NODE ID    ITER.  POWER(W) GBS      CPI      GFLOPS/W TIME(s)  AVG_F IMC_F IO(MBS) MPI%
      G-POWER(T/U) G-FREQ   G-UTIL(G/MEM)
 175966-1         cmp2506    21     360.6    115.8    0.838    0.086    1.001    2.58  2.30  0.0     11.6
      0.0   /   0.0  0.00     0%/0%
 175966-1         cmp2507    21     333.7    118.4    0.849    0.081    1.001    2.58  2.32  0.0     12.0
      0.0   /   0.0  0.00     0%/0%
 175966-1         cmp2506    31     388.6    142.3    1.010    0.121    1.113    2.58  2.38  0.0     19.7
      0.0   /   0.0  0.00     0%/0%
 175966-1         cmp2507    31     362.8    142.8    1.035    0.130    1.113    2.59  2.37  0.0     19.5
      0.0   /   0.0  0.00     0%/0%
 175966-1         cmp2506    41     383.3    143.2    1.034    0.124    1.114    2.58  2.38  0.0     19.6
      0.0   /   0.0  0.00     0%/0%
[user@host EAR]$ eacct -j 175966 -c test.csv
 Successfully written applications to csv. Only applications with EARL will have its information properly
      written.

[user@host EAR]$ eacct -j 175966.1 -c -l test.csv
 Successfully written applications to csv. Only applications with EARL will have its information properly
      written.
```

2.6 Job energy optimization: EARL policies

The core component of EAR at the user's job level is the EAR Library (EARL). The Library deals with job monitoring and is the component which implements and applies optimization policies based on monitored workload.

We highly recommend you to read EARL documentation and also how energy policies work in order to better understand what is doing the Library internally, so you will can explore easily all features (e.g., tunning variables, collecting data) EAR offers to the end-user so you will have more knowledge about how much resources your application consumes and how to correlate with its computational characteristics.

2.7 Data visualization with Grafana

EAR data can be visualized with Grafana dashboards in two different ways: Using grafana with SQL queries (depending on your Data Center configuration) and visualizing data collected with `eacct` and loading locally. The second option will be explained since you might expect to not having access to the EAR Database.

Once you have your own Grafana instance running, you need to install *csv-datasource*:

```
bin/grafana-cli plugins install marcusolsson-csv-datasource (You can first check if it's already available
      by testing the available Data sources)
```

Enable the CSV plug-in by creating a `custom.ini` file in the conf directory with the following content:

```
[plugin.marcusolsson-csv-datasource]
allow_local_mode = true
```

Once you have a local server running on your PC or laptop, open your web browser and connect to Grafana at the URL: `http://localhost:3000/login`. Next steps are:

Create the Data source

In the left menu, select Configuration/Data source/Add data source. Select CSV data source from the list of options. You need to create a new data source for each CSV file you are going to visualize. For each one, select *Local*. Note that the path must to be a **public directory**.

Import the Dashboard

Go to the left menu, *Dashboard*, and select the *Import* option. This option allows you uploading or selecting a json file with pre-specified graphs, tables, etc. Graphs are associated with data sources, so you may need to change the Data Source name in the json file to match the one you've created on Grafana. The json file is `here`, and below you can see the Data Source names expected. There is a configuration for two data sources: *EAR_loops* for visualizing CSV files containing EAR loop signatures (e.g., `eacct [-j <job_id>[.<step_id>]] -r -c <filename>`) and *EAR_app* for visualizing application signatures (e.g., `eacct [-j <job_id>[.<step_id>]] -l -c <filename>`).

```
{
  "__inputs": [
    {
      "name": "DS_EAR_LOOPS",
      "label": "EAR_loops",
      "description": "",
      "type": "datasource",
      "pluginId": "marcusolsson-csv-datasource",
      "pluginName": "CSV"
    },
    {
      "name": "DS_EAR_APPS",
      "label": "EAR_apps",
      "description": "",
      "type": "datasource",
      "pluginId": "marcusolsson-csv-datasource",
      "pluginName": "CSV"
    }
  ],
```

Import the JSON file to create the visualization dashboards and refresh the URL the browser page. Below you can see an example of what you will see.

Figure 2.1 EAR Grafana Dashboard example

Chapter 3

Tutorials

Quick links to EAR tutorials:

- ISC'24, Energy Management and Optimization with EAR.
- 6th MareNostrum Hackathon, Energy monitoring of HPC/AI workloads with EAR in MN5.

Chapter 4

EAR commands

EAR offers the following commands:

- Commands to analyze data stored in the DB: eacct and ereport.
- Commands to control and temporally modify cluster settings: econtrol.
- Commands to create/update/clean the DB: edb_create, edb_clean_pm and edb_clean_apps.
- A command to run OpenMPI applications with EAR on SLURM systems through `mpirun` command: erun.
- A command to show current EAR installation information: ear-info.

Commands belonging to the first three categories read the EAR configurarion file (`ear.conf`) to determine whether the user is authorized, as some of them has some features (or the wall command) only available that set of users. Root is a special case, it doesn't need to be included in the list of authorized users. Some options are disabled when the user is not authorized.

> **NOTE** EAR module must be loaded in your environment in order to use EAR commands.

4.1 EAR job Accounting (eacct)

The `eacct` command shows accounting information stored in the EAR DB for jobs (and step) IDs. The command uses EAR's configuration file to determine if the user running it is privileged or not, as **non-privileged users can only access their information**. It provides the following options.

```
Usage: eacct [Optional parameters]
    Optional parameters:
        -h  displays this message
        -v  displays current EAR version
        -b  verbose mode for debugging purposes
        -u  specifies the user whose applications will be retrieved. Only available to privileged users.
      [default: all users]
        -j  specifies the job id and step id to retrieve with the format [jobid.stepid] or the format
      [jobid1,jobid2,...,jobid_n].
                A user can only retrieve its own jobs unless said user is privileged. [default: all jobs]
        -a  specifies the application names that will be retrieved. [default: all app_ids]
        -c  specifies the file where the output will be stored in CSV format. If the argument is "no_file"
      the output will be printed to STDOUT [default: off]
        -t  specifies the energy_tag of the jobs that will be retrieved. [default: all tags].
        -s  specifies the minimum start time of the jobs that will be retrieved in YYYY-MM-DD. [default: no
      filter].
        -e  specifies the maximum end time of the jobs that will be retrieved in YYYY-MM-DD. [default: no
      filter].
        -l  shows the information for each node for each job instead of the global statistics for said job.
```

```
  -x  shows the last EAR events. Nodes, job ids, and step ids can be specified as if were showing job
information.
  -m  prints power signatures regardless of whether mpi signatures are available or not.
  -r  shows the EAR loop signatures. Nodes, job ids, and step ids can be specified as if were showing
job information.
  -o  modifies the -r option to also show the corresponding jobs. Should be used with -j.
  -n  specifies the number of jobs to be shown, starting from the most recent one. [default: 20][to
get all jobs use -n all]
  -f  specifies the file where the user-database can be found. If this option is used, the information
will be read from the file and not the database.
```

The basic usage of `eacct` retrieves the last 20 applications (by default) of the user executing it. If a user is **privileged**, they may see all users applications. The default behaviour shows data from each job-step, aggregating the values from each node in said job-step. If using SLURM as a job manager, a *sb* (sbatch) job-step is created with the data from the entire execution. A specific job may be specified with `-j` option.

Below table shows some examples of `eacct` usage.

Command line	Description
eacct	Shows last 20 jobs executed by the user.
eacct -j <JobID>	Shows data of the job <JobID>, one row for each step of the job.
eacct -j <JobID>.<StepID>	Shows data of the step <StepID> of job <JobID>.
eacct -j <JobIDx>,<JobIDy>,<JobIDz>	Shows data of jobs (one row per step) <JobIDx>,<JobIDy> and <JobIDz>.

The command shows a pre-selected set of columns:

Column field	Description
JOB-STEP	JobID and StepID reported. JobID-∗sb∗ is shown for the sbatch step in SLURM systems.
USER	The username of the user who executed the job.
APPLICATION	Job's name or executable name if job name is not provided.
POLICY	Energy optimization policy name. *MO* means for monitoring, *ME* for min_energy, *MT* for min_time and *NP* is the job ran without EARL.
NODES	Number of nodes involved in the job run.
AVG/DEF/IMC(GHz)	Average CPU frequency, default frequency and average uncore frequency. Includes all the nodes for the step. In GHz.
TIME(s)	Average step execution time along all nodes, in seconds.
POWER(W)	Average node power along all the nodes, in Watts.
GBS	CPU main memory bandwidth (GB/second). Hint for CPU/Memory bound classification.
CPI	CPU Cycles per Instruction. Hint for CPU/Memory bound classification.
ENERGY(J)	Accumulated node energy. Includes all the nodes. In Joules.
GFLOPS/W	CPU GFlops per Watt. Hint for energy efficiency. The metric uses the number of operations, not instructions.
IO(MBS)	I/O (read and write) Mega Bytes per second.
MPI%	Percentage of MPI time over the total execution time. It's the average including all the processes and nodes.

If EAR supports GPU monitoring/optimisation, the following columns are added:

Column field	Description
G-POW (T/U)	Average GPU power. Accumulated per node and average along involved nodes. *T* mean for total GPU power consumed (even the job is not using any or all of GPUs in one node). *U* means for only used GPUs on each node.
G-FREQ	Average GPU frequency. Per node and average of all the nodes.
G-UTIL(G/MEM)	GPU utilization and GPU memory utilization.

For node-specific information, the `-l` (i.e., long) option provides detailed accounting of each individual node. In addition, `eacct` shows an additional column: `VPI(%)`. The VPI is meaning the percentage of AVX512 instructions over the total number of instructions.

For runtime data (EAR loops) one may retrieve them with `-r`. Both Job and Step ID filtering works. To easily transfer command's output, `-c` option saves it in .csv format. Both aggregated and detailed accountings are available, as well as filtering:

Command line	Description
eacct -j <JobID> -c test.csv	Adds to the file `test.csv` all metrics shown above for each step if the job <JobID>.
eacct -j <JobID>.<StepID> -l -c test.csv	Appends to the file `test.csv` all metrics in the EAR DB for each node involved in step <StepID> of job <JobID>.
eacct -j <JobID>.<StepID> -r -c test.csv	Appends to the file `test.csv` all metrics in EAR DB for each loop of each node involved in step <StepID> of job <JobID>.

When requesting long format (i.e., `-l` option) or runtime metrics (i.e., `-r` option) to be stored in a CSV file (i.e., `-c` option), header names change from the output shown when you don't request CSV format. Below table shows header names of CSV file storing long information about jobs:

Field name	Description
NODENAME	The node name the row information belongs to.
JOBID	The JobID.
STEPID	The StepID. For the sbatch step, `SLURM_BATCH_SCRIPT` value is printed.
USERID	The username of the user who executed the job.
GROUPID	The group name of the user who executed the job.
JOBNAME	Job's name or executable name if job name is not provided.
USER_ACC	The account name of the user who executed the job.
ENERGY_TAG	The energy tag used if the user set one for its job step.
POLICY	Energy optimization policy name. *MO* means for monitoring, *ME* for min_energy, *MT* for min_time and *NP* is the job ran without EARL.
POLICY_TH	The policy threshold used by the optimization policy set with the job.
AVG_CPUFREQ_KHZ	Average CPU frequency of the job step executed in the node, expressed in kHz.
AVG_IMCFREQ_KHZ	Average uncore frequency of the job step executed in the node, expressed in kHz. **Default data fabric frequency on AMD sockets**.
DEF_FREQ_KHZ	default frequency of the job step executed in the node, expressed in kHz.
TIME_SEC	Execution time (in seconds) of the application in the node. As this is computed by EARL, *sbatch* step does not contain such info.
CPI	CPU Cycles per Instruction. Hint for CPU/Memory bound classification.
TPI	Memory transactions per Instruction. Hint for CPU/Memory bound classification.
MEM_GBS	CPU main memory bandwidth (GB/second). Hint for CPU/Memory bound classification.
IO_MBS	I/O (read and write) Mega Bytes per second.
PERC_MPI	Percentage of MPI time over the total execution time.
DC_NODE_POWER↩ _W	Average node power, in Watts.
DRAM_POWER_W	Average DRAM power, in Watts. **Not available on AMD sockets**.
PCK_POWER_W	Average RAPL package power, in Watts.
CYCLES	Total number of cycles.
INSTRUCTIONS	Total number of instructions.
CPU-GFLOPS	CPU GFlops per Watt. Hint for energy efficiency. The metric uses the number of operations, not instructions.

Field name	Description
L1_MISSES	Total number of L1 cache misses.
L2_MISSES	Total number of L2 cache misses.
L3_MISSES	Total number of L3/LLC cache misses.
SPOPS_SINGLE	Total number of single precision 64 bit floating point operations.
SPOPS_128	Total number of single precision 128 bit floating point operations.
SPOPS_256	Total number of single precision 256 bit floating point operations.
SPOPS_512	Total number of single precision 512 bit floating point operations.
DPOPS_SINGLE	Total number of double precision 64 bit floating point operations.
DPOPS_128	Total number of double precision 128 bit floating point operations.
DPOPS_256	Total number of double precision 256 bit floating point operations.
DPOPS_512	Total number of double precision 512 floating point 512 operations.

If EAR supports GPU monitoring/optimisation, the following columns are added:

Field name	Description
GPU∗x∗_POWER_W	Average GPU∗x∗ power, in Watts.
GPU∗x∗_FREQ_KHZ	Average GPU∗x∗ frequency, in kHz.
GPU∗x∗_MEM_FREQ_KHZ	Average GPu∗x∗ memory frequency, in kHz.
GPU∗x∗_UTIL_PERC	Average percentage of GPU∗x∗ utilization.
GPU∗x∗_MEM_UTIL_PERC	Average percentage of GPU∗x∗ memory utilization.

For runtime metrics (i.e., `-r` option), *USERID*, *GROUPID*, *JOBNAME*, *USER_ACC*, *ENERGY_TAG* (as energy tags disable EARL), *POLICY* and *POLICY_TH* are not stored at the CSV file. However, the iteration time (in seconds) is present on each loop as *ITER_TIME_SEC*, as well as a timestamp (i.e., *TIMESTAMP*) with the elapsed time in seconds since the EPOCH.

4.2 EAR system energy Report (ereport)

The ereport command creates reports from the energy accounting data from nodes stored in the EAR DB. It is intended to use for energy consumption analysis over a set period of time, with some additional (optional) criteria such as node name or username.

```
Usage: ereport [options]
Options are as follows:
        -s start_time           indicates the start of the period from which the energy consumed will be
      computed. Format: YYYY-MM-DD. Default: end_time minus insertion time*2.
        -e end_time             indicates the end of the period from which the energy consumed will be
      computed. Format: YYYY-MM-DD. Default: current time.
        -n node_name |all       indicates from which node the energy will be computed. Default: none (all
      nodes computed)
                                        'all' option shows all users individually, not aggregated.
        -u user_name |all       requests the energy consumed by a user in the selected period of time.
      Default: none (all users computed).
                                        'all' option shows all users individually, not aggregated.
        -t energy_tag|all       requests the energy consumed by energy tag in the selected period of time.
      Default: none (all tags computed).
                                        'all' option shows all tags individually, not aggregated.
        -i eardbd_name|all      indicates from which eardbd (island) the energy will be computed. Default:
      none (all islands computed)
                                        'all' option shows all eardbds individually, not aggregated.
        -g                      shows the contents of EAR's database Global_energy table. The default
      option will show the records for the two previous T2 periods of EARGM.
                                        This option can only be modified with -s, not -e
        -x                      shows the daemon events from -s to -e. If no time frame is specified, it
      shows the last 20 events.
        -v                      shows current EAR version.
        -h                      shows this message.
```

4.2.1 Examples

The following example uses the 'all' nodes option to display information for each node, as well as a start_time so it will give the accumulated energy from that moment until the current time.

```
[user@host EAR]$ ereport -n all -s 2018-09-18
     Energy (J)       Node      Avg. Power (W)
     20668697         node1        146
     20305667         node2        144
     20435720         node3        145
     20050422         node4        142
     20384664         node5        144
     20432626         node6        145
     18029624         node7        128
```

This example filters by EARDBD host (one per island typically) instead:

```
[user@host EAR]$ ereport -s 2019-05-19 -i all
     Energy (J)        Node
     9356791387        island1
    30475201705        island2
    37814151095        island3
    28573716711        island4
    29700149501        island5
    26342209716        island6
```

And to see the state of the cluster's energy budget (set by the sysadmin) you can use the following:

```
[user@host EAR]$ ereport -g
Energy%  Warning lvl            Timestamp       INC th      p_state    ENERGY T1    ENERGY T2      TIME T1
      TIME T2        LIMIT       POLICY
111.486          100  2019-05-22 10:31:34            0          100          893      1011400       907200
      600       604800 EnergyBudget
111.492          100  2019-05-22 10:21:34            0          100          859      1011456       907200
      600       604800 EnergyBudget
111.501          100  2019-05-22 10:11:34            0          100          862      1011533       907200
      600       604800 EnergyBudget
111.514          100  2019-05-22 10:01:34            0          100          842      1011658       907200
      600       604800 EnergyBudget
111.532          100  2019-05-22 09:51:34            0          100          828      1011817       907200
      600       604800 EnergyBudget
111.554            0  2019-05-22 09:41:34            0            0          837      1012019       907200
      600       604800 EnergyBudget
```

4.2.2 EAR Control (econtrol)

The `econtrol` command modifies cluster settings (temporally) related to power policy settings. These options are sent to all the nodes in the cluster.

> **NOTE** Any changes done with `econtrol` will not be reflected in `ear.conf` and thus will be lost when reloading the system.

```
Usage: econtrol [options]
        --status                                ->requests the current status for all nodes. The ones
      responding show the current
                                                        power, IP address and policy configuration. A list
      with the ones not
                                                        responding is provided with their hostnames and IP
      address.
                                                        --status=node_name retrieves the status of that node
      individually.
        --type          [status_type]           ->specifies what type of status will be requested: hardware,
                                                        policy, full (hardware+policy), app_node,
      app_master, eardbd, eargm or power. [default:hardware]
        --power                                 ->requests the current power for the cluster.
                                                        --power=node_name retrieves the current power of
      that node individually.
        --set-freq      [newfreq]               ->sets the frequency of all nodes to the requested one
        --set-def-freq  [newfreq]  [pol_name]   ->sets the default frequency for the selected policy
        --set-max-freq  [newfreq]               ->sets the maximum frequency
        --set-powercap  [new_cap]               ->sets the powercap of all nodes to the given value. A node
      can be specified
                                                         after the value to only target said node.
        --hosts         [hostlist]              ->sends the command only to the specified hosts. Only works
      with status, power_status,
                                                         --power and --set-powercap
        --restore-conf                          ->restores the configuration for all nodes
```

```
  --active-only                    ->supresses inactive nodes from the output in hardware
status.
  --health-check                   ->checks all EARDs and EARDBDs for errors and prints all
that are unresponsive.
  --mail [address]                 ->sends the output of the program to address.
  --ping                           ->pings all nodes to check whether the nodes are up or not.
Additionally,
                                           --ping=node_name pings that node individually.
  --version                        ->displays current EAR version.
  --help                           ->displays this message.
```

econtrol's status is a useful tool to monitor the nodes in a cluster. The most basic usage is the hardware status (default type) which shows basic information of all the nodes.

```
[user@login]$ econtrol --status
hostname      power   temp    freq    job_id  stepid
   node2        278    66C    2.59      6878       0
   node3        274    57C    2.59      6878       0
   node4         52    31C    1.69         0       0

INACTIVE NODES
node1   192.0.0.1
```

The application status type can be used to retrieve all currently running jobs in the cluster. app_master gives a summary of all the running applications while app_node gives detailed information of each node currently running a job.

```
[user@login]$ econtrol --status --type=app_master
Job-Step    Nodes   DC power      CPI      GBS   Gflops     Time Avg Freq
  6878-0        2     280.13     0.37    24.39   137.57    54.00     2.59

[user@login]$ econtrol --status --type=app_node
Node id     Job-Step   M-Rank   DC power      CPI      GBS   Gflops     Time Avg Freq
  node2       6878-0        0     280.13     0.37    24.39   137.57    56.00     2.59
  node3       6878-0        1     245.44     0.37    24.29   136.40    56.00     2.59
```

4.3 Database commands

4.3.1 edb_create

Creates the EAR DB used for accounting and for the global energy control. Requires root access to the MySQL server. It reads the ear.conf to get connection details (server IP and port), DB name (which may or may not have been previously created) and EAR's default users (which will be created or altered to have the necessary privileges on EAR's database).

```
Usage:edb_create [options]
        -p       Specify the password for MySQL's root user.
        -o       Outputs the commands that would run.
        -r       Runs the program. If '-o' this option will be override.
        -h       Shows this message.
```

4.3.2 edb_clean_pm

Cleans periodic metrics from the database. Used to reduce the size of EAR's database, it will remove every Periodic_metrics entry older than num_days:

```
Usage:./src/commands/edb_clean_pm [options]
    -d num_days     REQUIRED: Specify how many days will be kept in database. (defaut: 0 days).
    -p          Specify the password for MySQL's root user.
    -o          Print the query instead of running it (default: off).
    -r          Execute the query (default: on).
    -h          Display this message.
    -v          Show current EAR version.
```

4.3.3 edb_clean_apps

Removes applications from the database. It is intended to remove old applications to speed up queries and free up space. It can also be used to remove specific applications from database. It removes ALL the information related to those jobs (the following tables will be modified for each job: Loops, if they exist; GPU_signatures, if they exist; Signatures, if they exist; Power signatures, Applications, and Jobs).

It is recommended to run the application with the -o option first to ensure that the queries that will be executed are correct.

```
Usage:edb_clean_apps [-j/-d] [options]
    -p      The program will request the database user's password.
    -u user     Database user to execute the operation (it needs DELETE privileges). [default: root]
    -j jobid.stepid Job id and step id to delete. If no step_id is introduced, every step within the job
      will be deleted
    -d ndays    Days to preserve. It will delete any jobs older than ndays.
    -o      Prints out the queries that would be executed. Exclusive with -r. [default:on]
    -r      Runs the queries that would be executed. Exclusive with -o. [default:off]
    -l      Deletes Loops and its Signatures. [default:off]
    -a      Deletes Applications and related tables. [default:off]
    -h      Displays this message
```

4.4 erun

`erun` is a program that simulates all the SLURM and EAR SLURM Plug-in pipeline. It was designed to provide compatibility between MPI implementations not fully compatible with SLURM SPANK plug-in mechanism (e.g., OpenMPI), which is used to set up EAR at job submission. You can launch `erun` with the `--program` option to specify the application name and arguments. See the usage below:

```
> erun --help

This is the list of ERUN parameters:
Usage: ./erun [OPTIONS]

Options:
    --job-id=<arg>  Set the JOB_ID.
    --nodes=<arg>   Sets the number of nodes.
    --program=<arg> Sets the program to run.
    --clean     Removes the internal files.

SLURM options:
...
```

The syntax to run an MPI application with `erun` has the form 'mpirun -n <X> erun --program='my_app arg1 arg2 .. argN'. Therefore, mpirun will run *X* erun processes. Then, erun will launch the application my_app` with the arguments passed, if specified. You can use as many parameters as you want but the semicolons have to cover all of them in case there are more than just the program name.

`erun` will simulate on the remote node both the local and remote pipelines for all created processes. It has an internal system to avoid repeating functions that are executed just one time per job or node, like SLURM does with its plugins.

IMPORTANT NOTE If you are going to launch `n` applications with `erun` command through a sbatch job, you must set the environment variable `SLURM_STEP_ID` to values from `0` to `n-1` before each `mpirun` call. By this way `erun` will inform the EARD the correct step ID to be stored then to the Database.

The `--job-id` and `--nodes` parameters create the environment variables that SLURM would have created automatically, because it is possible that your application make use of them. The `--clean` option removes the temporal files created to synchronize all ERUN processes.

Also you have to load the EAR environment module or define its environment variables in your environment or script:

Variable	Parameter
EAR_INSTALL_PATH=<path>	prefix=<path>
EAR_TMP=<path>	localstatedir=<path>
EAR_ETC=<path>	sysconfdir=<path>
EAR_DEFAULT=<on/off>	default=<on/off>

4.5 ear-info

`ear-info` is a tool created to quickly view useful information about the current EAR installation of the system. It shows relevant details for both users and administrators, such as configuration defaults, installation paths, etc.

```
[user@hostname ~]$ ear-info -h
Usage: ear-info [options]
    --node-conf[=nodename]
    --help
```

The tool prints out information without giving it any argument. It shows a resume about EAR parameters set at compile time, as well as some installation dependent configuration:

- The current EAR version.
- The maximum number of CPUs/processors supported.
- The maximum number of sockets supported.
- Whether the current installation provides support for GPUs.
- The default optimization policy.
- Whether the EAR Library is enabled by default on job submission.
- Information about EAR's Uncore Frequency Scaling policy (eUFS) configuration.
- EAR's dynamic load balancing policy.
- EAR's application phase classification.
- EAR's MPI stats collection feature.
- EAR data reporting mechanism configuration.

Below there is an example of the output:

```
EAR version 4.3
Max CPUS supported set to 256
Max sockets supported set to 4
EAR installed with GPU support MAX_GPUS 8
Default cluster policy is monitoring
EAR optimization  by default set to 0


Environment configuration section..............
                                  eUFS  1
                              eUFS limit 0.02
                    Load balanced enabled 1
                         Load Balance th 0.80
             Use turbo for critical path 1
                               Use turbo 0
                           Exclusive mode 0
                          Use EARL phases 1
                        Use energy models 1
     Max IMC frequency (0 = not defined) 0
     Min IMC frequency (0 = not defined) 0
 GPU frequency/pstate (0 = max GPU freq) 0
                          MPI optimization 0
                            MPI statistics 0
                               App. Tracer no trace
              App. Extra report plugins no extra plugins
            App. reporting loops to EARD 1
.........................................


HACK section...........................
                            Install path /hpc/base/ctt/packages/EAR/ear
               Energy optimization policy
                         GPU power policy
      /hpc/base/ctt/packages/EAR/ear/lib/plugins/policies/gpu_monitoring.so
                         CPU power  model
      /hpc/base/ctt/packages/EAR/ear/lib/plugins/policies/gpu_monitoring.so
                  CPU shared power  model
      /hpc/base/ctt/packages/EAR/ear/lib/plugins/models/cpu_power_model_default.so
.........................................
```

EAR was designed to be installed on heterogeneous systems, so there are some configuration parameters that are applied to a set of nodes identified by different tags. The `--node-conf` flag can be used to request additional information about a specific node. Configuration related to EAR's power capping sub-system, default optimization policies configuration and other parameters associated with the node requested are retrieved. You can read the EAR configuration section for more details about how EAR uses tags to identify and configure different kind of nodes on a given heterogeneous system.

Contact with `ear-support@bsc.es` for more information about the nomenclature used by `ear-info`'s output.

Chapter 5

Environment variables

5.1 Introduction

EAR offers some environment variables in order to provide users the opportunity to tune or request some of EAR features. They must be exported before the job submission, e.g., in the batch script.

The current EAR version has support for SLURM, PBS and OAR batch schedulers. In SLURM systems the scheduler may filter environment variables not prefixed with *SLURM_* character set (this happens when the batch script is submitted purging all environment variables to work in a clean environment). For that reason, the first design of EAR environment variables was to have variable names with the form *SLURM_*<variable_name>.

Now that EAR has support for other batch schedulers, and in order to maintain the coherency of environment variables names, below environment variables need the prefix of the scheduler used on the system the job is submitted on, plus an underscore. For example, in SLURM systems, the environment variable presented as EAR_↩LOADER_APPLICATION must be exported as SLURM_EAR_LOADER_APPLICATION in the submission batch script. In an OAR installed system, this variable would be exported as OAR_EAR_LOADER_APPLICATION. This design may only have a real effect on SLURM systems, but it makes it easier for the development team to provide support for multiple batch schedulers.

All examples showing the usage of below environment variables assume a system using SLURM.

5.2 Loading EAR Library

5.2.1 EAR_LOADER_APPLICATION

Rules the EAR Loader to load the EAR Library for a specific application that does not follow any of the current programming models (or maybe a sequential app) supported by EAR. Your system must have installed the non-MPI version of the Library (ask your system administrator).

The value of the environment variable must coincide with the job name of the application you want to launch with EAR. If you don't provide it, the EAR Loader will compare it against the executable name. For example:

```
#!/bin/bash

export EAR_LOADER_APPLICATION=my_job_name

srun --ntasks 1 --job-name=my_job_name ./my_exec_file
```

See the Use cases section to read more information about how to run jobs with EAR.

5.2.2 EAR_LOAD_MPI_VERSION

Forces to load a specific MPI version of the EAR Library. This is needed, for example, when you want to load the EAR Library for Python + MPI applications, where the Loader is not able to detect the MPI implementation the application is going to use. Accepted values are either *intel* or *open mpi*. The following example runs Tensorflow 1 benchmarks for several convulational neural networks with EAR. It can be downloaded from Tensorflow benchmarks repository.

```
#!/bin/bash

#SBATCH --job-name=TensorFlow
#SBATCH -N 8
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=18

# Specific modules here
# ...

export EAR_LOAD_MPI_VERSION="open mpi"

srun --ear-policy=min_time \
    python benchmarks/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
    ... more application options
```

See the Use cases section to read more information about how to run jobs with EAR.

5.3 Report plug-ins

5.3.1 EAR_REPORT_ADD

Specify a report plug-in to be loaded. The value must be a shared object file, and it must be located at `$EAR_INSTALL_PATH/lib/plugins/report` or at the path from where the job was launched. Alternatively, you can provide the full path (absolute or relative) of the report plug-in.

```
#!/bin/bash

export EAR_REPORT_ADD=my_report_plugin.so:my_report_plugin2.so

srun -n 10 my_mpi_app
```

For more information, see [Report plug-ins](Report) section.

5.3.2 Extended GPU metrics

EAR library collects a set of extra GPU metrics on demand. To enable and report it two env var has to be defined `EAR_GPU_DCGMI_ENABLED` to enable them, the dcgmi.so report plugin has to be requested, and the `DCGMI_LOGS_PATH` has to be set to specify the path for csv generated. This last env var is optional, if not set, the current working directory is used. By default, only few metrics are gathered, to collect all of them set the `EAR_DCGM_ALL_EVENTS` env var.

5.4 Verbosity

5.4.1 EARL_VERBOSE_PATH

Specify a path to create a file (one per node involved in a job) where to print messages from the EAR Library. This is useful when you run a job in multiple nodes, as EAR verbose information for each of them can result in lots of messages mixed at stderr (EAR messages default channel). Also, there are applications that print information in both stdout and stderr, so maybe a user wants to have information separated.

If the path does not exist, EAR will create it. The format of generated files names is `earl_log.<node_rank>.<local_rank>.<job_step>.<job_id>`, where the *node_rank* is an integer set by EAR from 0 to *n_nodes - 1* involved in the job, and it indicates to which node the information belongs to. The local rank is an arbitrary rank set by EAR of a process in the node (from 0 to *n_procceses_in_node - 1*). It indicates which process is printing messages to the files, and it will be always the first one indexed, i.e., 0. Finally, the *job_step* and *job_id* are fields showing information about the job corresponding to the execution from where messages were generated.

```
#!/bin/bash

#SBATCH -j my_job_name
#SBATCH -N 2
#SBATCH -n 96
export EARL_VERBOSE_PATH=ear_logs_dir_name
export I_MPI_HYDRA_BOOTSTRAP=slurm
export I_MPI_HYDRA_BOOTSTRAP_EXEC_EXTRA_ARGS="--ear-verbose=1"

mpirun -np 96 -ppn 48 my_app
```

After the above job example completion, in the same directory where the application was submitted, there will be a directory called *ear_logs_dir_name* with two files, i.e., one for each node, called *earl_logs.0.0.<job_step>.<job_id>* and *earl_logs.1.0.<job_step>.<job_id>*, respectively.

5.5 Frequency management

5.5.1 EAR_GPU_DEF_FREQ

Set a GPU frequency (in kHz) to be fixed while your job is running. The same frequency is set for all GPUs used by the job.

```
#!/bin/bash

#SBATCH -J gromacs-cuda
#SBATCH -N 1
export I_MPI_PIN=1
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi2.so
input_path=/hpc/appl/biology/GROMACS/examples
input_file=ion_channel.tpr
GROMACS_INPUT=$input_path/$input_file
export EAR_GPU_DEF_FREQ=1440000

srun --cpu-bind=core --ear-policy=min_energy gmx_mpi mdrun \
    -s $GROMACS_INPUT -noconfout -ntomp 1
```

5.5.2 EAR_JOB_EXCLUSIVE_MODE

Indicate whether the job will run in a node exclusively (non-zero value). EAR will reduce the CPU frequency of those cores not used by the job. This feature explodes a very easy vector of power saving.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -n 64
#SBATCH --cpus-per-task=2
#SBATCH --exclusive

export EAR_JOB_EXCLUSIVE_MODE=1

srun -n 10 --ear=on ./mpi_mpi_app
```

5.5.3 Controlling Uncore/Infinity Fabric frequency

EARL offers the possibility to control the Integrated Memory Controller (IMC) for Intel(R) architectures and Infinity Fabric (IF) for AMD architectures. On this page we will use the term *uncore* to refer both of them. Environment variables related to uncore control covers policy specific settings or the chance for a user to fix it during an entire job.

5.5.3.1 EAR_SET_IMCFREQ

Enables/disables EAR's [eUFS](publications) feature. Type `ear-info` to see whehter eUFS is enabled by default.

You can control eUFS' maximum permitted time penalty by exporting `EAR_POLICY_IMC_TH`, which is a float indicating the threshold value that prevents the policy to reduce so much the uncore frequency, possible leading to considerable performance penalty.

Below example enables eUFS with a penalty threshold of 3.5%:

```
#!/bin/bash
...

export SLURM_EAR_SET_IMCFREQ=1
export SLURM_EAR_POLICY_IMC_TH=0.035
...

srun [...] my_app
```

5.5.3.2 EAR_MAX_IMCFREQ and EAR_MIN_IMCFREQ

Set the maximum and minimum values (in kHz) at which *uncore* frequency should be. Two variables were designed because Intel(R) architectures let to set a range of frequencies that limits its internal UFS mechanism. If you set both variables with different values, the minimum one will be set.

Below example shows a job execution fixing the uncore frequency at 2.0GHz:

```
#!/bin/bash
...

export EAR_MAX_IMCFREQ=2000000
export EAR_MIN_IMCFREQ=2000000
...

srun [...] my_app
```

5.5.4 Load Balancing

By default, EAR policies try to set the best CPU (and uncore, if enabled) frequency according to node grain metrics. This behaviour can be changed telling EAR to detect and deal with unbalanced workloads, i.e., there is no equity between processes regarding their MPI/computational activity.

When EAR detects such behaviour, policies slightly modify its way of CPU frequency selection by setting a different frequency for each process' cores according how far it is from the critical path. Please, contact with `ear-support@bsc.es` if you want more details about how it works.

> A correct CPU binding it's required to get the most benefit of this feature. Check the documentation of you application programming model/vendor/flavour or yur system batch scheduler.

5.5.4.1 EAR_LOAD_BALANCE

Enables/Disables EAR's Load Balance strategy in energy policies. Type `ear-info` to see whether this feature is enabled by default.

Load unbalance detection algorithm is based on `POP-CoE`'s Load Balance Efficiency metric, which is computed as the ratio between average useful computation time (across all processes) and maximum useful computation time (also across all processes). By default (if `EAR_LOAD_BALANCE` is enabled), a node load balance efficiency below **0.8** will trigger EAR's Load Balancing algorithm. This threshold value can be modified by setting `EAR_LOAD_BALANCE_TH` environment variable. For example, if you want to be more permissive with the application load balance and prevent per-process CPU frequency selection, you can increase the load balance threshold:

```
#!/bin/bash
...

export EAR_LOAD_BALANCE=1
export EAR_LOAD_BALANCE_TH=0.89
...

srun [...] my_app
```

5.5.5 Support for Intel(R) Speed Select Technology

Since version 4.2, EAR supports the interaction with `Intel(R) Speed Select Technology (Intel(R) SST)` which lets the user to have more fine grained control over per-CPU Turbo frequency. This feature opens a door to users for getting more control over the performance (also power consumption) across CPUs running their applications and jobs. It is available on selected SKUs of Intel(R) Xeon(R) Scalable processors. For more information about Intel(R) SST, below are listed useful links to official documentation:

- `Intel(R) SST-CP`
- `Intel(R) SST-TF`
- `The Linux Kernel: Intel(R) Speed Select Technology User Guide`

EAR offers two environment variables that let to specify a list of priorities (CLOS) in two different ways. The first one will set a CLOS for each task involved in the job. On the other hand, the second offered variable will set a list of priorities per CPU involved in the job. Values must be within the range of available CLOS that Intel(R) SST provides you.

If some of the two supported environment variables are set, EAR will set-up all of its internals transparently if the architecture supports it. Also, it will restore configuration on the job ending. If Intel(R) SST is not supported, no effect will occur. If you enable EARL verbosity you will see the mapping of the CLOS set for each CPU in the node. Note that a `-1` value means that no change was done on the specific CPU.

5.5.5.1 EAR_PRIO_TASKS

A list that specifies the CLOS that CPUs assigned to tasks must be set. This variable is useful because you can configure your application transparently without concerning about the affinity mask that the scheduler is assigning to your tasks. You can use this variable when you know (or guess) your application's tasks workload and you want to tune it by setting manually different Turbo priorities. Note that you still need to ensure that different tasks do not share CPUs.

For example, imagine you want to submit a job that runs a MPI application with 16 tasks, each one pinned on a single core, in a two-socket Intel(R) Xeon(R) Platinum 8352Y with 32 cores each, with Hyper-threading enabled, i.e., each task will run on two CPUs and 32 of the total 128 will be allocated by this application. Below could be a (simplified) batch script that submits this example:

```
#!/bin/bash

export EAR_PRIO_TASKS=0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3

srun --ntasks=16 --cpu-bind=core,verbose --ear-policy=monitoring --ear-cpufreq=2201000 --ear-verbose=1
      bin/bt.C.x
```

The above script sets CLOS 0 to tasks 0 to 3, CLOS 1 to tasks 4 to 7, CLOS 2 to tasks 8 to 11 and CLOS 3 to tasks 12 to 15. The `srun` command binds each task to one core (through `--cpu-bind` flag), sets the turbo frequency and enables EAR verbosity. Below there is the output message shown by the batch scheduler (i.e., SLURM):

```
cpu-bind=MASK - ice2745, task  0  0 [23363]: mask 0x10000000000000001 set
cpu-bind=MASK - ice2745, task  1  1 [23364]: mask 0x10000000000000000100000000 set
cpu-bind=MASK - ice2745, task  2  2 [23365]: mask 0x20000000000000002 set
cpu-bind=MASK - ice2745, task  3  3 [23366]: mask 0x20000000000000000200000000 set
cpu-bind=MASK - ice2745, task  4  4 [23367]: mask 0x40000000000000004 set
cpu-bind=MASK - ice2745, task  5  5 [23368]: mask 0x40000000000000000400000000 set
cpu-bind=MASK - ice2745, task  6  6 [23369]: mask 0x80000000000000008 set
cpu-bind=MASK - ice2745, task  7  7 [23370]: mask 0x80000000000000000800000000 set
cpu-bind=MASK - ice2745, task  8  8 [23371]: mask 0x100000000000000010 set
cpu-bind=MASK - ice2745, task  9  9 [23372]: mask 0x10000000000000001000000000 set
cpu-bind=MASK - ice2745, task 10 10 [23373]: mask 0x200000000000000020 set
cpu-bind=MASK - ice2745, task 11 11 [23374]: mask 0x20000000000000002000000000 set
cpu-bind=MASK - ice2745, task 12 12 [23375]: mask 0x400000000000000040 set
cpu-bind=MASK - ice2745, task 13 13 [23376]: mask 0x40000000000000004000000000 set
cpu-bind=MASK - ice2745, task 14 14 [23377]: mask 0x800000000000000080 set
cpu-bind=MASK - ice2745, task 15 15 [23378]: mask 0x80000000000000008000000000 set
```

We can see here that SLURM spreaded out tasks accross the two sockets of the node, e.g., task 0 runs on CPUs 0 and 64, task 1 runs on CPUs 32 and 96. Below output shows how EAR sets and verboses CLOS list per CPU in the node. Following the same example, you can see that CPUs 0, 64, 32 and 96 have priority/CLOS 0. Note that those CPUs not involved in the job show a -1.

```
Setting user-provided CPU priorities...
PRIO0: MAX GHZ - 0.0 GHz (high)
PRIO1: MAX GHZ - 0.0 GHz (high)
PRIO2: MAX GHZ - 0.0 GHz (low)
PRIO3: MAX GHZ - 0.0 GHz (low)
[000, 0] [001, 0] [002, 1] [003, 1] [004, 2] [005, 2] [006, 3] [007, 3]
[008,-1] [009,-1] [010,-1] [011,-1] [012,-1] [013,-1] [014,-1] [015,-1]
[016,-1] [017,-1] [018,-1] [019,-1] [020,-1] [021,-1] [022,-1] [023,-1]
[024,-1] [025,-1] [026,-1] [027,-1] [028,-1] [029,-1] [030,-1] [031,-1]
[032, 0] [033, 0] [034, 1] [035, 1] [036, 2] [037, 2] [038, 3] [039, 3]
[040,-1] [041,-1] [042,-1] [043,-1] [044,-1] [045,-1] [046,-1] [047,-1]
[048,-1] [049,-1] [050,-1] [051,-1] [052,-1] [053,-1] [054,-1] [055,-1]
[056,-1] [057,-1] [058,-1] [059,-1] [060,-1] [061,-1] [062,-1] [063,-1]
[064, 0] [065, 0] [066, 1] [067, 1] [068, 2] [069, 2] [070, 3] [071, 3]
[072,-1] [073,-1] [074,-1] [075,-1] [076,-1] [077,-1] [078,-1] [079,-1]
[080,-1] [081,-1] [082,-1] [083,-1] [084,-1] [085,-1] [086,-1] [087,-1]
[088,-1] [089,-1] [090,-1] [091,-1] [092,-1] [093,-1] [094,-1] [095,-1]
[096, 0] [097, 0] [098, 1] [099, 1] [100, 2] [101, 2] [102, 3] [103, 3]
[104,-1] [105,-1] [106,-1] [107,-1] [108,-1] [109,-1] [110,-1] [111,-1]
[112,-1] [113,-1] [114,-1] [115,-1] [116,-1] [117,-1] [118,-1] [119,-1]
[120,-1] [121,-1] [122,-1] [123,-1] [124,-1] [125,-1] [126,-1] [127,-1]
```

5.5.5.2 EAR_PRIO_CPUS

A list of priorities that should have the same length as the number of CPUs your job is using. This configuration lets to set up CPUs CLOS in a more low level way: **the *n-th* priority value of the list will set the priority of the *n-th* CPU your job is using.**

This way of configuring priorities rules the user to know exactly the affinity of its job's tasks before launching the application, so it becomes harder to use if your goal is the same as the one you can get by setting the above environment variable: task-focused CLOS setting. But it becomes more flexible when the user has more control over the affinity set to its application, because you can discriminate between different CPUs assigned to the same task. Moreover, this is the only way to set different priorities over different threads in no-MPI applications.

5.5.6 EAR_MIN_CPUFREQ

This variable can only be set by **authorized users**, and modifies the minimum CPU frequency the EAR Library can set. The EAR configuration file has a field called `cpu_max_pstate` which sets this limits on the tag it is configured. Authorized users can modify this limit at submission time by using this environment to test, for example, the best value for the `ear.conf` field.

5.5.7 Disabling EAR's affinity masks usage

For both [Load Balancing](load-balancing) and Intel(R) SST support, EAR uses processes' affinity mask read at the beginning of the job. If you are working on an application that changes (or may change) the affinty mask of tasks dynamically, this can lead some miss configuration not detected by EAR. To avoid any unexpected problem, **we highly recommend you** to export `EARL_NO_AFFINITY_MASK` environment variable **even you are not planning to work with some of the mentioned features**.

Note: Since EAR version 5.0, EAR updates the process mask periodically (aprox 1 sec.) and always before applying the optimization policy.

5.6 Workflow support

5.6.1 EAR_DISABLE_NODE_METRICS

By defining this environment variable, the user or workflow manager indicates EAR the current process must not be considered as power consumer, not affecting the CPU power models used to estimate the amount of power corresponding to each application sharing a node. This env variable target master-worker scenarios (or map-reduce) when one process is not doing computational work, just working as master creating and waiting for processes. By specifying this var, the EARL ignores the affinity mas for this process and assumes its activity is not relevant for the whole power consumption. The value is not relevant, it only has to be defined. In a fork-join program (or similar, it has to be unset before the creation of the workers.

5.6.2 EAR_NTASK_WORK_SHARING

By defining this environment variable, the user indicates the library the set of processes sharing the node are in fact a single application (not MPI). This enables a synchronization at the beginning and all the processes with same jobid and stepid (or similar for other schedulers different than SLURM) works together. Only one of the will be selected as the master and will apply the energy policy. For GPU applications it's mandatory the process can access all the GPUs. Otherwise, it is not recommended and each process will apply its own optimization. The value is not relevant, it only has to be defined. **This feature is only supported on systems using SLURM**.

5.7 Data gathering/reporting

5.7.1 EARL_REPORT_LOOPS

Since **version 4.3**, EAR can be configured to not report application loop signatures by default. This configuration satisfy a constraint for many HPC data centers where hundreds of jobs are launched daily, leading to too many loops reported and a quick EAR database size increase.

For those users which still want to get application loop data, this variable can be set to one (i.e., `export EARL↩_REPORT_LOOPS=1`) to force EAR report their application loop signatures. Therefore, users can get their loop data by calling `eacct -j <job_id> -r`.

5.7.2 EAR_GET_MPI_STATS

Use this variable to generate two files at the end of the job execution that will contain global, per process MPI information. You must specify the prefix (optionally with a path) of the filename. One file (*[path/]prefix.ear_mpi_stats.↩full_nodename.csv*) will contain a resume about MPI throughput (per-process), while the other one (*[path/]prefix.↩ear_mpi_calls_stats.full_nodename.csv*) will contain a more fine grained information about different MPI call types. Here is an example:

```
!#/bin/bash

#SBATCH -j mpi_job_name
#SBATCH -n 48

MPI_INFO_DST=$SLURM_JOBID-mpi_stats
mkdir $MPI_INFO_DST

export EAR_GET_MPI_STATS=$MPI_INFO_DST/$SLURM_JOB_NAME

srun -n 48 --ear=on ./mpi_app
```

At the end of the job, two files will be created at the directory named `\<job_id\>-mpi_stats` located in the same directory where the application was submitted. They will be named *mpi_job_name.ear_mpi_stats.full_nodename.csv* and *mpi_job_name.ear_mpi_calls_stats.full_nodename.csv*. File pairs will be created for each node involved in the job.

Take into account that each process appends its own MPI statistics to files. This behavior does not guarantee that the header of files will be on the first line of them, as only one process writes it. You must move it at the top of each file manually before reading them with some tool you use to visualize and work with CSV files, e.g., spreadsheet, a R or Python package.

Below table shows fields available by **ear_mpi_stats** file:

Field	Description
mrank	The EAR's internal node ID used to identify the node.
lrank	The EAR's internal rank ID used to identify the process.
total_mpi_calls	The total number of MPI calls.
exec_time	The execution time, in microseconds.
mpi_time	The time spent in MPI calls, in microseconds.
perc_mpi_time	The percentage of total execution time (i.e., *exec_time*) spent in MPI calls.

Below table shows fields available by **ear_mpi_calls_stats** file:

Field	Description
Master	The EAR's internal node ID used to identify the node.
Rank	The EAR's internal rank ID used to identify the process.
Total MPI calls	The total number of MPI calls.
MPI_time/Exec_time	The ration between time spent in MPI calls and the total execution time.
Exec_time	The execution time, in microseconds.
Sync_time	Time spent (in microseconds) in **blocking** synchronization calls, i.e., MPI_Wait, MPI_Waitall, MPI_Waitany, MPI_Waitsome and MPI_Barrier.
Block_time	Time spent in blocking calls, i.e., MPI_Allgather, MPI_Allgatherv, MPI_Allreduce, MPI_Alltoall, MPI_Alltoallv, MPI_Barrier, MPI_Bcast, MPI_Bsend, MPI_Cart_create, MPI_Gather, MPI_Gatherv, MPI_Recv, MPI_Reduce, MPI_Reduce_scatter, MPI_Rsend, MPI_Scan, MPI_Scatter, MPI_Scatterv, MPI_Send, MPI_Sendrecv, MPI_Sendrecv_replace, MPI_Ssend and all *Wait* calls of **Sync_time** field.
Collec_time	Time spent in **blocking** collective calls, i.e., MPI_Allreduce, MPI_Reduce and MPI_Reduce_scatter.
Total MPI sync calls	Total number of synchronization calls.
Total blocking calls	Total number of blocking calls.
Total collective calls	Total number of collective calls.
Gather	Total number of **blocking** Gather calls, i.e., MPI_Allgather, MPI_Allgatherv, MPI_Gather and MPI_Gatherv.
Reduce	Total number of **blocking** Reduce calls, i.e., MPI_Allreduce, MPI_Reduce and MPI_Reduce_scatter.
All2all	Total number of **blocking** All2all calls, i.e., MPI_Alltoall and MPI_Alltoallv.
Barrier	Total number of **blocking** Barrier calls, i.e., MPI_Barrier.
Bcast	Total number of **blocking** Bcast calls, i.e., MPI_Bcast.
Send	Total number of **blocking** Send calls, i.e., MPI_Bsend, MPI_Rsend, MPI_Send and MPI_Ssend.
Comm	Total number of **blocking** Comm calls, i.e., MPI_Cart_create.
Receive	Total number of **blocking** Receive calls, i.e., MPI_Recv.
Scan	Total number of **blocking** Scan calls, i.e., MPI_Scan.

Field	Description
Scatter	Total number of **blocking** Scatter calls, i.e., MPI_Scatter and MPI_Scatterv.
SendRecv	Total number of **blocking** SendRecv calls, i.e., MPI_Sendrecv, MPI_Sendrecv_replace.
Wait	Total number of **blocking** Wait calls, i.e., all MPI_Wait calls.
t_Gather	Time (in microseconds) spent in **blocking** Gather calls.
t_Reduce	Time (in microseconds) spent in **blocking** Reduce calls.
t_All2all	Time (in microseconds) spent in **blocking** All2all calls.
t_Barrier	Time (in microseconds) spent in **blocking** Barrier calls.
t_Bcast	Time (in microseconds) spent in **blocking** Bcast calls.
t_Send	Time (in microseconds) spent in **blocking** Send calls.
t_Comm	Time (in microseconds) spent in **blocking** Comm calls.
t_Receive	Time (in microseconds) spent in **blocking** Receive calls.
t_Scan	Time (in microseconds) spent in **blocking** Scan calls.
t_Scatter	Time (in microseconds) spent in **blocking** Scatter calls.
t_SendRecv	Time (in microseconds) spent in **blocking** SendRecv calls.
t_Wait	Time (in microseconds) spent in **blocking** Wait calls.

5.7.3 EAR_TRACE_PLUGIN

EAR offers the chance to generate Paraver traces to visualize runtime metrics with the `Paraver tool`. Paraver is a visualization tool developed by CEPBA-Tools team and currently maintained by the Barcelona Supercomputing Center's tools team.

The EAR trace generation mechanism was designed to support different trace generation plug-ins although the Paraver trace plug-in is the only supported by now. You must set the value of this variable to `tracer_paraver.so` to load the tracer. This shared object comes with the official EAR distribution and it is located at `$EAR_INSTALL_PATH/lib/plugins/tracer`. Then you need to set the `EAR_TRACE_PATH` variable (see below) to specify the destination path of the generated Paraver traces.

5.7.4 EAR_TRACE_PATH

Specify the path where you want to store the trace files generated by the EAR Library. The path must be fully created. Otherwise, the Paraver tracer plug-in won't be loaded.

Here is an example of the usage of the above explained environment variables:

```
#!/bin/bash
...

export EAR_TRACE_PLUGIN=tracer_paraver.so
export EAR_TRACE_PATH=$(pwd)/traces
mkdir -p $EAR_TRACE_PATH
srun -n 10 --ear=on ./mpi_app
```

5.7.5 REPORT_EARL_EVENTS

Use this variable (i.e., `export REPORT_EARL_EVENTS=1`) to make EARL send internal events to the [Database](EAR-Database). These events are useful to have more information about Library's behaviour, like when DynAIS **(REFERENCE DYNAIS)** is turned off, the computational phase EAR is guessing the application is on or the status of the applied policy **(REF POLICIES)**. You can query job-specific events through `eacct -j <JobID> -x`, and you will get a table of all reported events:

Field name	Description
Event_ID	Internal ID of the event stored at the Database.
Timestamp	*yyyy-mm-dd hh:mm:ss*.
Event_type	Which kind of event is it. Possible event types explained below.
Job_id	The JobID of the event.
Value	The value stored with the event. Categorical events explained below.
node_id	The node from where the event was reported.

5.7.5.1 Event types

Below are listed all kind of event types you can get when requesting job events. For categorical event values, the (value, category) mapping is explained.

- **policy_error** Reported when the policy couldn't select the optimal frequency.
- **dynais_off** Reported when DynAIS is turned off and the Library becomes in *periodic monitoring mode*.
- **earl_state** The internal EARL state. Possible values are:
  - **0** This is the initial state and stands for no iteration detected.
  - **1** EAR starts computing the signature
  - **2** EAR computes the local signature and executes the per-node policy.
  - **3** This state computes a new signature and evaluates the accuracy of the policy.
  - **4** Projection error.
  - **5** This is a transition state to recompute EARL timings just in case we need to adapt it because of the frequency selection.
  - **6** Signature has changed.
- **optim_accuracy** The internal optimization policy state. Possible values are:
  - **0** Policy not ready.
  - **1** Policy says all is ok.
  - **2** Policy says it'n not ok.
  - **3** Policy wants to try again to optimize.

The above event types may be useful only for advanced users. Please, contact with ear-support@bsc.es if you want to know more about EARL internals.

- **energy_saving** Energy (in %) EAR is guessing the policy is saving.
- **power_saving** Power in (in %) EAR is guessing the policy is saving.
- **performance_penalty** Execution time (in %) EAR is guessing the policy is incrementing.

Chapter 6

Admin guide

6.1 EAR Components

EAR is composed of five main components:

- **Node Manager (EARD):** It is a Linux service which provides the basic node power monitoring and job accounting. It also offers an API to be used for third-parties (e.g., other EAR components) to to make priviledged operations. It must have root access to the node (usually all compute nodes) where it will be running.
- **Database Manager (EARDBD):** A Linux service (it normally runs in a service node) which caches data to be stored in a database reducing the number of queries. We currently support MariaDB and Postgres↩SQL. This compoment is not needed to be enabled/used if don't use such database services to report EAR data.
- **Global Manager (EARGM):** A Linux service (it normally runs in a service node) which provides cluster-level support (e.g., powercap). It needs access to all nodes where a Node Manager is runningi the cluster.
- **EAR Library (EARL):** A Job Manager (distributed as a shared object) which provides job/application -level monitoring and optimization.
- **SLURM plug-in:** A SLURM SPANK plug-in which provides support for using EAR job accounting and loading EARL transparently for users on systems using SLURM.

The following image shows the main interactions between components:

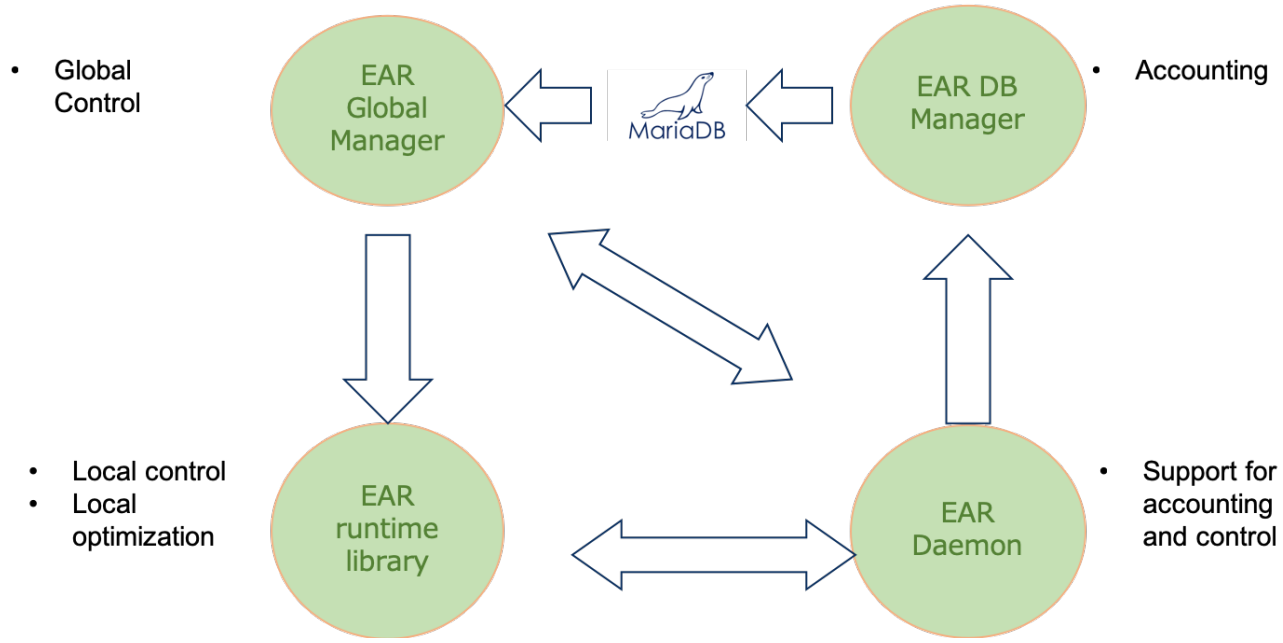


Figure 6.1 EAR components diagram

For a more detailed information about EAR components, visit the Architecture page.

6.2 Quick Installation Guide

This section provides summary of needed steps for compiling and installing EAR. The complete guide is left into [another section](Installation from source), but **it is recommended to read first this section** since it contains useful information about how and what gets compiled and installed.

Check out first whether your system satisfies all [requirements](EAR-requirements), then check that you have Autoconf version 2.69 or later. You can then bootstrap the build system:

```
autoreconf -i
```

As commented in the overview, the EAR Library might be loaded along with MPI applications thanks to the EAR Loader library. The latter detects the application symbols at runtime and loads the right Library. Therefore, you should compile at least two versions of the EAR Library:

- One for MPI applications (using one of the `MPI implementations supported`.
- One for non-MPI applications.

This is an example to configure EAR to be compiled for both versions:

```
# Configure EAR to compile the non-MPI version of the EAR Library
./configure --disable-mpi \
    MPICC=mpicc MAKE_NAME=nompi

# Configure EAR to compile the MPI version of the Library
./configure MPICC=mpicc MPICC_FLAGS="-O2 -g" MAKE_NAME=impi
```

> **The above example assumes your MPI Library is Intel MPI.** If you want to compile EARL for another MPI flavour check out [this section](supporting-more-than-one-mpi-implementation).

EAR currently does not support GNU make parallel builds, so the above example must be run in the source code root directory. For the same reason, the `configure` script support a variable called `MAKE_NAME`, so it generates a Makefile called `Makefile.<MAKE_NAME variable value>`. Therefore you can call `make` program targeting each configuration Makefile generated program targeting each configuration Makefile generated.

The flag `--disable-mpi` is used for configuring the non-MPI version of the EAR Library.

> **Note** that even when configuring for this use case, `MPICC` variable is also set. This is because the EAR Loader still needs MPI headers for checking whether the application being running is MPI, and `configure` finds out these by checking this variable.

After completing the previous steps, you can compile and install EAR by targeting each of the generated Makefiles. the following example takes the Makefile suffixes used in the previous one:

```
# Compile and install EAR. The EAR Library version installed
# will be for supporting non-MPI applications.
make -f Makefile.nompi
make -f Makefile.nompi install

# sysconfdir installation needs another target
make -f Makefile.nompi etc.install
make -f Makefile.nompi doc.install

# Compile and install just the
# MPI version of the EAR Library
make -f Makefile.impi full
make -f Makefile.impi earl.install
```

In the above example, some non-standard targets are used. `etc.install` target is needed for installing all configuration, module and service files to be used later when configuring EAR. The `full` target is the equivalent of calling first `clean` and then `all` targets. Finally, `earl.install` is used for installing the EAR Library, since we are just compiling again because we want another version of the Library installed along with the previous one.

EAR Makefiles include a specific target for each component, supporting full or partial updates:

Target	Description
`install`	Reinstalls all the files except `etc` and `doc`.
`earl.install`	Reinstalls only the EARL.
`eard.install`	Reinstalls only the EARD.
`earplug.install`	Reinstalls only the EAR SLURM plugin.
`eardbd.install`	Reinstalls only the EARDBD.
`eargmd.install`	Reinstalls only the EARGMD.
`reports.install`	Reinstalls only report plugins.

Here is an example of a bash script summarizing the information provided until now, compiling and installing EAR with two versions of the Library: one supporting Intel MPI applications and other one supporting any non-MPI application:

```
#!/bin/bash

# This script bootstraps, configures, compiles and installs
# EAR with two versions of the Library: one supporting Intel MPI applications
# and other one supporting any non-MPI application
#
# Requirements:
# - GNU Autoconf
# - GNU make
# - A modern C compiler
# - Intel MPI compiler and Library

EAR_INSTALL_PATH= # Set the root location of your installation
EAR_TMP= # Set the location of temporary directories and files
EAR_ETC= # Set the location of configuration and services files.

my_CFLAGS="-O2 -g"

# Bootstrap the configure script
```

```
autoreconf -i

# Configure EAR to compile the non-MPI version of the EAR Library
./configure --disable-mpi --prefix=$EAR_INSTALL_PATH \
    MPICC=mpicc CC=gcc CC_FLAGS="$my_CFLAGS" \
  EAR_TMP=$EAR_TMP EAR_ETC=$EAR_ETC \
  MAKE_NAME=nompi

# Compile and install EAR. The EAR Library version installed
# will be for supporting non-MPI applications.
make -f Makefile.nompi
make -f Makefile.nompi install

# sysconfdir installation needs another target
make -f Makefile.nompi etc.install
make -f Makefile.nompi doc.install

# Configure EAR to compile the MPI version of the Library.
./configure --prefix=$EAR_INSTALL_PATH \
    MPICC=mpicc MPICC_FLAGS="$my_CFLAGS" \
    CC=gcc CC_FLAGS="$my_CFLAGS" \
    EAR_TMP=$EAR_TMP EAR_ETC=$EAR_ETC \
    MAKE_NAME=impi

# Compile and install just the
# MPI version of the EAR Library
make -f Makefile.impi full
make -f Makefile.impi earl.install
```

After compiling and installing following the previous step, you should have the following directories under `configure`'s `--prefix` flag used path:

- `bin`: Including commands and tools.
- `sbin`: Includes EAR services binaries.
- `etc`: Includes templates and examples for EAR service files, the `ear.conf` file, the EAR module and so.
- `lib`: Includes all libraries and plugins.
- `include`
- `man`: Man pages.

Inside `lib` directory, apart from plug-ins, you should see at least three files.

- `libearld.so`: This is the EAR Loader.
- `libear.so`: This is the EAR Library compiled with Intel MPI symbols. See the [next section](supporting-more-than-one-mpi-implementation) if you need support for other MPI implementations.
- `libear.gen.so`: This is the EAR Library compiled without MPI symbols. The `.gen` extension is added automatically when setting `--disable-mpi` flag.

6.2.1 Supporting more than one MPI implementation

Many systems have different MPI implementations installed, so users can choose which one fits better for their applications. Even all of them provide the same interface, each one has some specific symbols not specified in the standard. Therefore you need to install an EAR Library version for each MPI flavor you want to support.

In order to help the EAR Loader to load the proper Library version, coliving libraries must be named different. This is accomplished by providing `MPI_VERSION` variable to `configure`. This variable sets an extension of the `libear.so` shared object compiled, so when the EAR Loader detects the MPI version of the application, it can easily load the proper Library. You need to set a specific value to variable value depending on the MPI implementation you are going to compile following this table:

Implementation	MPI_VERSION value	EARL Name
Intel MPI	not required	libear.so (default)
MVAPICH	not required	libear.so (default)
OpenMPI	ompi	libear.ompi.so
Fujitsu MPI	fujitsu	libear.fujitsu.so
Cray MPI	cray	libear.cray.so

Note that in the example used until now this variable was not used. This is because for this MPI version the EAR Loader does not find for an extension, and it is the continuation of the first EARL design and it was not changed.

So, if you would like to add to your previous EAR installation the support for, let's say, OpenMPI, you should type the following:

```
# Configure EAR to compile Library supporting OpenMPI applications
# Note: mpicc must point to an OpenMPI installation
./configure MPICC=mpicc MPICC_FLAGS="-O2 -g" MAKE_NAME=openmpi MPI_VERSION=ompi

make -f Makefile.openmpi full
# The below line assumes you already have installed all other components,
# i.e., `make -f Makefile.<extension> install`.
make -f Makefile.openmpi earl.install
```

This is an example of a bash script which summarizes the configuration, compilation and installation of EAR providing support for multiple MPI implementations:

```
#!/bin/bash

# This script bootstraps, configures, compiles and installs
# EAR with two versions of the Library: one supporting Intel MPI applications
# and other one supporting any non-MPI application
#
# Requirements:
# - GNU Autoconf
# - GNU make
# - A modern C compiler
# - Intel MPI compiler and Library

EAR_INSTALL_PATH= # Set the root location of your installation
EAR_TMP= # Set the location of temporary directories and files
EAR_ETC= # Set the location of configuration and services files.

my_CFLAGS="-O2 -g"

# Bootstrap the configure script
autoreconf -i

# Replace with an Intel MPI module
module load intel-mpi-module

# Configure EAR to compile the non-MPI version of the EAR Library
./configure --disable-mpi --prefix=$EAR_INSTALL_PATH \
    MPICC=mpicc CC=gcc CC_FLAGS="$my_CFLAGS" \
  EAR_TMP=$EAR_TMP EAR_ETC=$EAR_ETC \
  MAKE_NAME=nompi

# Compile and install EAR. The EAR Library version installed
# will be for supporting non-MPI applications.
make -f Makefile.nompi
make -f Makefile.nompi install

# sysconfdir installation needs another target
make -f Makefile.nompi etc.install
make -f Makefile.nompi doc.install

# Configure EAR to compile the MPI version of the Library.
./configure --prefix=$EAR_INSTALL_PATH \
    MPICC=mpicc MPICC_FLAGS="$my_CFLAGS" \
    CC=gcc CC_FLAGS="$my_CFLAGS" \
    EAR_TMP=$EAR_TMP EAR_ETC=$EAR_ETC \
    MAKE_NAME=impi

# Compile and install just the
# MPI version of the EAR Library
make -f Makefile.impi full
make -f Makefile.impi earl.install

# Configure EAR to compile Library supporting OpenMPI applications
# Note: mpicc must point to an OpenMPI installation
module unload intel-mpi-module
```

```
module load openmpi-module

./configure --prefix=$EAR_INSTALL_PATH \
    MPICC=mpicc MPICC_FLAGS="$my_CFLAGS" \
    CC=gcc CC_FLAGS="$my_CFLAGS" \
    EAR_TMP=$EAR_TMP EAR_ETC=$EAR_ETC \
    MAKE_NAME=openmpi MPI_VERSION=ompi

make -f Makefile.openmpi full
make -f Makefile.openmpi earl.install
```

6.2.2 Deployment and validation

6.2.2.1 Monitoring: Compute node and DB

Prepare the configuration

Either installing from sources or rpm, EAR installs a template for `ear.conf` file in `$EAR_ETC/ear/ear.conf.template` and `$EAR_ETC/ear/ear.conf.full.template`. The full version includes all fields. Copy only one as `$EAR_ETC/ear/ear.conf` and update with the desired configuration. Go to the configuration section to see how to do it. The `ear.conf` is used by all the services. It is recommended to have in a shared folder to simplify the changes in the configuration.

EAR module

Install and load EAR module to enable commands. It can be found at `$EAR_ETC/module`. You can add ear module whan it is not in standard path by doing `module use $EAR_ETC/module` and then `module load ear`.

EAR Database

Create EAR database with `edb_create`, installed at `$EAR_INSTALL_PATH/sbin`. The `edb_create -p` command will ask you for the DB root password. If you get any problem here, check first whether the node where you are running the command can connect to the DB server. In case problems persists, execute `edb_create -o` to report the specific SQL queries generated. In case of trouble, contact with `ear-support@bsc.es` or open in issue.

Energy models

EAR uses a power and performance model based on systems signatures. These system signatures are stored in coefficient files.

Before starting EARD, and just for testing, it is needed to create a dummy coefficient file and copy in the coefficients path, by default placed at`$EAR_ETC/coeffs`. Use the `coeffs_null` application from tools section.

> EAR version 4.1 does not require null coefficients.

EAR services

Create soft links or copy EAR service files to start/stop services using system commands such as `systemctl` in the services folder. EAR service files are generated at `$EAR_ETC/systemd` and they can usually be placed in `$(ETC)/systemd`.

- EARD must be started on compute nodes.
- EARDBD must be started on service nodes (can be any node with DB access).

Enable and start EARDs and EARDBDs via services (e.g., `sudo systemctl start eard`, `sudo systemctl start eardbd`). EARDBD and EARD outputs can be found at `$EAR_TMP/eardbd.server.log` and `$EAR_TMP/eard.log` respectively when *DBDaemonUseLog* and *NodeUseLog* options are set to *1* in the `ear.conf` file, respectively. Otherwise, their outputs are generated at *stderr* and can be seen using the `journalctl` command (i.e., journalctl -u eard).

By default, a certain level of verbosity is set. It is not recommended to modify it but you can change it by modifying the value of constants in file `src/common/output/output_conf.h`.

Quick validation

Check that EARDs are up and running correctly with `econtrol --status` (note that daemons will take around a minute to correctly report energy and not show up as an error in econtrol). EARDs create a per-node text file with values reported to the EARDBD (local to compute nodes). In case there are problems when running econtrol, you can also find this file at `$EAR_TMP/nodename.pm_periodic_data.txt`.

Check that EARDs are reporting metrics to database with ereport. `ereport -n all` should report the total energy sent by each daemon since the setup.

6.2.2.2 Monitoring: EAR plugin

- Set up EAR's SLURM plugin (see the configuration section for more information).

  It is recommented to create a soft link to the `$EAR_ETC/slurm/ear.plugstack.conf` file in the `/etc/slurm/plugstack.conf.d` directory to simplify the EAR plugin management.

  For a first test it is recommened to set `default=off` in the `ear.plugstack.conf` (to disable the automatic loading of the EAR library).

EAR plugin validation

At this point you must be able to see EAR options when doing, for example, `srun --help`. You must see something like below as part of the output. The EAR plugin must be enabled at login and compute nodes.

```
[user@hostname ~]$ srun --help
Usage: srun [OPTIONS(0)... [executable(0) [args(0)...]]] [ : [OPTIONS(N)...]] executable(N) [args(N)...]

Parallel run options:
...

Constraint options:
...

Consumable resources related options:
...

Affinity/Multi-core options: (when the task/affinity plugin is enabled)
...

Options provided by plugins:
      --ear=on|off            Enables/disables Energy Aware Runtime Library
      --ear-policy=type       Selects an energy policy for EAR
                              {type=default,gpu_monitoring,monitoring,min_energ-
                              y,min_time,gpu_min_energy,gpu_min_time}
      --ear-cpufreq=frequency Specifies the start frequency to be used by EAR
                              policy (in KHz)
      --ear-policy-th=value   Specifies the threshold to be used by EAR policy
                              (max 2 decimals) {value=[0..1]}
      --ear-user-db=file      Specifies the file to save the user applications
                              metrics summary 'file.nodename.csv' file will be
                              created per node. If not defined, these files
                              won't be generated.
      --ear-verbose=value     Specifies the level of the
                              verbosity{value=[0..1]}; default is 0
      --ear-learning=value    Enables the learning phase for a given P_STATE
                              {value=[1..n]}
      --ear-tag=tag           Sets an energy tag (max 32 chars)

...

Help options:
  -h, --help                  show this help message
      --usage                 display brief usage message

Other options:
  -V, --version               output version information and exit
```

- Submit one application via SLURM and check that it is correctly reported to the database with `eacct` command.

  Note that only privileged users can check other users' applications.

- Submit one MPI application (corresponding with the version you have compiled) with `--ear=on` and check that now the output of `eacct` includes the Library metrics.
- Set `default=on` to set the EAR Library loading by default at `ear.plugstack.conf`. If default is turned off, EARL can be explicitly loaded by setting the flag `--ear=off` at job submission.

At this point, you can use EAR for monitoring and accounting purposes but it cannot use the power policies offered by EARL. To enable them, first perform a learning phase and compute node coefficients. See the EAR learning phase wiki page. For the coefficients to be active, restart daemons.

> **Important** Reloading daemons will NOT make them load coefficients, restarting the service is the only way.

6.3 Installing from RPM

EAR includes the specification files to create an rpm **from an already existing installation**. Once created, it can be included in the compute nodes images. It is recommened only when no more changes are expected on the installation or when your compute fleet has ephimeral storage and EAR is installed in a non-shared file system.

The spec file is placed at `etc/rpms/specs/ear.spec` and it is generated from `etc/rpms/specs/ear.spec.in` at configuration time. The RPM can be part of the system image. Visit the Requirements page for a quick overview of the requirements.

Execute the `rpmbuild.sh` script to create the EAR rpm file. This is script is located at `etc/rpms` and it is created from `etc/rpms/rpmbuild.sh.in` at configuration time. **Run it from its location**. The rpm file will be located at `$HOME/rpmbuild/RPMS`. You can install it by typing:

```
rpm -ivh <ear_rpm_filename>.rpm
```

> You can also use the `--nodeps` if your dependency test fails. Type `rpm -e <ear_rpm_filename>` to uninstall.

6.3.1 Installation content

The `*.in` configuration files are compiled into `etc/ear/ear.conf.template` and `etc/ear/ear.full.conf.template`, `etc/module/ear`, `etc/slurm/ear.plugstack.conf` and various `etc/systemd/ear*.service`. You can find more information in the configuration page. Below table describes the complet heriarchy of the EAR installation:

Directory	Content / description
`/usr/lib`	Libraries and the scheduler plugin.
`/usr/lib/plugins`	EAR plugins.
`/usr/bin`	EAR commands.
`/usr/bin/tools`	EAR tools for coefficients computation.
`/usr/sbin`	Privileged components: EARD, EARDBD, EARGMD.
`/etc/ear`	Configuration files templates.
`/etc/ear/coeffs`	Folder to store coefficient files.
`/etc/module`	EAR module.
`/etc/slurm`	EAR SLURM plugin configuration file.
`/etc/systemd`	EAR service files.

6.3.2 RPM requirements

EAR uses some third party libraries. EAR RPM will not ask for them when installing but they must be available in `LD_LIBRARY_PATH` when running an application and you want to use EAR. Depending on the RPM, different version must be required for these libraries:

Library	Minimum version	References
MPI	-	-
MySQL*	15.1	MySQL or MariaDB
PostgreSQL*	9.2	PostgreSQL
Autoconf	2.69	Website
GSL	1.4	Website

- Just one of them required.

These libraries are not required, but can be used to get additional functionality or metrics:

Library	Minimum version	References
SLURM	17.02.6	Website
PBS**	2021	PBSPro or OpenPBS
CUDA/NVML	7.5	CUDA
CUPTI**	7.5	CUDA
Likwid	5.2.1	Likwid
FreeIPMI	1.6.8	FreeIPMI
OneAPI/L0**	1.7.9	OneAPI
LibRedFish**	1.3.6	LibRedFish

** These will be available in next release.

Also, some **drivers** has to be present and loaded in the system when starting EAR:

Driver	File	Kernel version	References
CPUFreq	kernel/drivers/cpufreq/acpi-cpufreq.ko	3.10	Information
Open IPMI	kernel/drivers/char/ipmi/*.ko	3.10	Information

6.4 Starting Services

The best way to execute all EAR daemon components (EARD, EARDBD, EARGM) is by the unit services method.

> **NOTE** EAR uses a MariaDB/MySQL server. The server must be started before EAR services are executed.

The way to launch the EAR daemons is via unit services. The generated unit services for the EAR Daemon, EAR Global Manager Daemon and EAR Database Daemon are generated and installed in `$(EAR_ETC)/systemd`. You have to copy those unit service files to your `systemd` operating system folder and then use the `systemctl` command to run the daemons. Check the EARD, EARDBD, EARGMD pages to find the precise execution commands.

When using `systemctl` commands, you can check messages reported to `stderr` using `journalctl`. For instance: `journalctl -u eard -f`. Note that if `NodeUseLog` is set to 1 in `ear.conf`, the messages will not be printed to `stderr` but to `$EAR_TMP/eard.log` instead. `DBDaemonUseLog` and `Globalmanager↩ UseLog` options in `ear.conf` specifies the output for EARDBD and EARGM, respectivelly.

Additionally, services can be started, stopped or reloaded on parallel using parallel commands such as `pdsh`. As an example: `sudo pdsh -w nodelist systemctl start eard`.

6.5 Updating EAR with a new installation

In some cases, it might be a good idea to create a new install instead of updating your current one, like trying new configurations or when a big update is released.

The steps to do so are:

- Install EAR in the new folder
- Replicate old etc (including `ear.conf` and coefficients) in the new one and update `ear.conf` with the new ETC path and whatever changes may be needed.
- Update EAR services in `/etc/systemd/system` folder (or equivalent, depending on your OS). Service files include ETC path and the absolute path for binaries.
- Update `/etc/slurm/plugstag.conf` with the new paths.
- Create a new EAR module with the updated paths.

Once all that is done, one should have two complete EAR installs that can be switched by changing the binaries that are executed by the services and changing the path in `plugstag.conf`.

6.6 Next steps

For a better overview of the installation process, return to the installation guide. To continue the installation, visit the configuration page to set up properly the EAR configuration file and the EAR SLURM plugin stack file.

Chapter 7

EAR-requirements

This section lists both software and harware requirements for compiling, running and using EAR. There is also a list of system requirements to use all EAR components and features.

7.1 Architectures

7.1.1 CPUs

Intel CPU families

- Skylake.
- IceLake.
- Sapphire Rappids.

AMD CPUs

- EPYC Rome, Milan and Genoa families.

Other

- ARM and Cray architectures are not tested in production.

7.1.2 GPUs

NVIDIA

- From Turing to Hopper.

Intel

- PVC.

7.2 Operating systems

EAR has been tested in CentOS (>=7), SUSE, Rocky and Red Hat Linux distributions.

7.3 Network

In the case you plan to report data to a database through the EARDBD, service nodes (wherever EARDBDs run) must be reachable by compute nodes, so EARDs can connect with them and report telemetry data. Moreover, EARDBDs and log-in nodes must be able to connect with Database servers for storing and retrieving data, respectively.

In order to be able to use administration commands, log-in nodes must be able to reach compute nodes.

7.3.1 Ports

- 1 TCP port for EARD on each compute node.
- 3 TCP ports for each EARDBD on service nodes.
- 1 TCP port for each EARGMD.

7.4 Compilation

You need at least a modern **C compiler**, **Autoconf** (>= 2.69) and **GNU make**. The rest of requirements are optional based on features you want to enable.

A **MPI compiler and headers** are nedded for supporting MPI applications. Intel MPI, OpenMPI, MVAPICH, Fujistsu and Cray MPICH are the versions currently supported.

If you want to retrieve NVIDIA GPU metrics as well as modify GPU clock frequency, you need **CUDA**. Check out the minimum required version based on your device. **OneAPI** (>= 1.7.9) for supporting the same features on Intel PVC GPUs.

SLURM must also be present if the SLURM SPANK plug-in wants to be used. Since current EAR version only supports automatic execution of applications with EAR Library using the SLURM plug-in, it must be running when EARL wants to be used (not needed for the most basic node monitoring service). EAR needs **slurm.h** and **spank.h** header files in this case.

EAR currently supports two relational databases for storing data. **MySQL** (>= 15.1) or **PostgreSQL** headers and libraries are nedded.

7.5 Energy and power readings

Your compute nodes should support one of these commands:

- **ipmitool dcmi power reading**.
- **ipmi-oem intelnm get-node-manager-statistics mode=globalpower**.
- **Lenovo SD650** commands for energy readings.
- Energy readings for **Intel Node Manager**.
- **freeipmi**.
- **libredfish**.

7.6 Kernel drivers

You system should have a Linux CPUFreq driver supporting *userspace* governor.

- **acpi_cpufreq** (recommended).
- **intel_cpufreq** and **intel_pstate** already tested and supported.

IPMI drivers must be installed in compute nodes. MSR kernel module must be loaded in compute nodes (**msr-safe** supported). Performance counters must be enabled (**perf** must be installed).

7.7 NVIDIA GPU metrics

The **nvidia-ml** library is the component used by EAR for reading NVIDIA GPU metrics. For devices prior to Hopper, **NVIDIA DCGM**'s *dcgmi* command is also required if you want more metrics than the GPU power, frequency and utilization and the GPu memory frequency and utilization.

7.8 AMD system management features

AMD HSMP module is needed for supporting a set of system management features.

7.9 Performance counters

Counters such as cycles, instructions, cache misses or FLOPS should be supported at user level. **perfparanoid** level should be set accordingly.

7.10 Learning phase

In order to compute coefficients during the learning phase, EAR comes with a set of tools which need **GSL** (>=1.4).

Chapter 8

Installation from source

8.1 Requirements

EAR requires some third party libraries and headers to compile and run, in addition to the basic requirements such as the compiler and Autoconf. This is a list of these **libraries**, minimum **tested** versions and its references:

Library	Minimum version	References
MPI	-	-
MySQL*	15.1	MySQL or MariaDB
PostgreSQL*	9.2	PostgreSQL
Autoconf	2.69	Website
GSL	1.4	Website

- Just one of them required.

These libraries are not required, but can be used to get additional functionality or metrics:

Library	Minimum version	References
SLURM	17.02.6	Website
PBS**	2021	PBSPro or OpenPBS
CUDA/NVML	7.5	CUDA
CUPTI**	7.5	CUDA
Likwid	5.2.1	Likwid
FreeIPMI	1.6.8	FreeIPMI
OneAPI/L0**	1.7.9	OneAPI
LibRedFish**	1.3.6	LibRedFish

** These will be available in next release.

Also, some **drivers** has to be present and loaded in the system:

Driver	File	Kernel version	References
CPUFreq	kernel/drivers/cpufreq/acpi-cpufreq.ko	3.10	Information
Open IPMI	kernel/drivers/char/ipmi/*.ko	3.10	Information

Lastly, the **compilers**: EAR uses C compilers. It has been tested with both Intel and GNU.

Compiler	Comment	Minimum version	References
GNU Compiler Collection (GCC)	For the library and daemon	4.8.5	Website
Intel C Compiler (ICC)	For the library and daemon	17.0.1	Website

8.2 Compilation and installation guide summary

1. Before the installation, make sure the installation path is accessible by all the computing nodes. Do the same in the folder where you want to set the configuration files (it will be called `$(EAR_ETC)` in this guide for simplicity).
2. Generate Autoconf's `configure` program by typing `autoreconf -i`.
3. Read sections below to understand how to properly set the `configure` parameters.
4. Compile EAR components by typing `./configure ...`, `make` and `make install` in the root directory.
5. Type `make etc.install` to install the content of `$(EAR_ETC)`. It is the configuration content, but that configuration will be expanded in the next section. You have a link at the bottom of this page.

8.3 Configure options

`configure` is based on shell variables which initial value could be given by setting variables in the command line, or in the environment. Take a look to the table with the most popular variables:

Variable	Description
MPICC	MPI compiler.
CC	C compiler command.
MPICC_FLAGS	MPI compiler flags.
CFLAGS	C compiler flags.
CC_FLAGS	Also C compiler flags.
LDFLAGS	Linker flags. E.g. '-L<lib dir>' if you have libraries in a nonstandard directory <lib dir>.
LIBS	Libraries to pass to the linker. E.g. '-l<library>'.
EAR_TMP	Defines the node local storage as 'var', 'tmp' or other tempfs file system (default: /var/ear) (you can alo use –localstatedir=DIR).
EAR_ETC	Defines the read-only single-machine data as 'etc' (default: EPREFIX/etc) (you can also use –sharedstatedir=DIR).
MAN	Defines the manual directory (default: PREFIX/man) (you can use also –mandir=DIR).
DOC	Defines the documentation directory (default: PREFIX/doc) (you can use also –docdir=DIR).
MPI_VERSION	Adds a suffix to the compiled EAR library name. Read further down this page for more information.
USER	Owner user of the installed files.
GROUP	Owned group of the installed files
MAKE_NAME	It adds an additional Makefile with a suffix.

- This is an example of `CC`, `CFLAGS` and `DEBUG` variables overwriting:
  `./configure CC=icc CFLAGS=-g EAR_ETC=/hpc/opt/etc`

You can choose the root folder by typing `./configure --PREFIX=<path>`. But there are other options in the following table:

Definition	Default directory	Content / description
<*PREFIX*>	/usr/local	Installation path
<*EAR_ETC*>	<*PREFIX*>/etc	Configuration files.
<*EAR_TMP*>	/var/ear	Pipes and temporal files.

You have more installation options information by typing `./configure --help`. If you want to change the value of any of this options after the configuration process, you can edit the root Makefile. All the options are at the top of the text and its names are self-explanatory.

Adding required libraries installed in custom locations

The `configure` script is capable to find libraries located in custom location if a module is loaded in the environment or its path is included in `LD_LIBRARY_PATH`. If not, you can help `configure` to find SLURM, or other required libraries in case you installed in a custom location. It is necessary to add its root path for the compiler to see include headers and libraries for the linker. You can do this by adding to it the following arguments:

Argument	Description
–with-cuda=<path>	Specifies the path to CUDA installation.
–with-freeipmi=<path>	Specify path to FREEIPMI installation.
–with-gsl=<path>	Specifies the path to GSL installation.
–with-likwid=<path>	Specifies the path to LIKWID installation.
–with-mysql=<path>	Specify path to MySQL installation.
–with-pgsql=<path>	Specify path to PostgreSQL installation.
–with-pbs	Enable PBS components.
–with-slurm=<path>	Specifies the path to SLURM installation.

- This is an example of `CC` overwriting the CUDA path specification:
  `./configure --with-cuda=/path/to/CUDA`

If unusual procedures must be done to compile the package, please try to figure out how `configure` could check whether to do them and contact the team to be considered for the next release. In the meantime, you can overwrite shell variables or export its paths to the environment (e.g. LD_LIBRARY).

Additional configure flags

Also, there are additional flags to help administrator increase the compatibility of EAR in nodes.

Argument	Description
–disable-rpath	Disables the RPATH included in binaries to specify some dependencies location.
–disable-avx512	Replaces the AVX-512 function calls by AVX-2.
–disable-gpus	The GPU monitoring data is not allocated nor inserted in the database.
–disable-mpi	Compiles the non-mpi version of the library.

8.4 Pre-installation fast tweaks

Some EAR characteristics can be modified by changing the value of the constants defined in `src/common/config/config_def.h`. You can open it with an editor and modify those pre-procesor variables to alter the EAR behaviour.

Also, you can quickly switch the user/group of your installation files by modifying the `CHOWN_USR/CHOWN_GRP` variables in the root Makefile.

8.5 Library distributions/versions

As commented in the overview, the EAR library is loaded next to the user MPI application by the EAR Loader. The library uses MPI symbols, so it is compiled by using the includes provided by your MPI distribution. The selection of the library version is automatic in runtime, but in the compiling and installation process is not required. Each compiled library has its own file name that has to be defined by the `MPI_VERSION` variable during `./configure` or by editing the root Makefile. The name list per distribution is exposed in the following table:

Distribution	Name	MPI_VERSION variable
Intel MPI	libear.so (default)	it is not required
MVAPICH	libear.so (default)	it is not required
OpenMPI	libear.ompi.so	ompi

If different MPI distributions shares the same library name, it means that its symbols are compatible between them, so compiling and installing the library one time will be enough. However, if you provide different MPI distributions to the users, you will have to compile and install the library multiple times.

Before compiling new libraries you have to install by typing `make install`. Then you can run the `./configure` again, changing the `MPICC`, `MPICC_FLAGS` and `MPI_VERSION` variables, or just opening the root `Makefile` and edit the same variables and `MPI_BASE`, which just sets the MPI installation root path. Now type `make full` to perform a clean compilation and `make earl.install`, to install only the new version of the library.

If your MPI version is not fully compatible, please contact `ear-support@bsc.es`. We will add compatibility to EAR and give you a solution in the meantime.

8.6 Other useful flags

You can install individual components by doing: `make eard.install` to install EAR Daemon, `make earl.install` to install EAR Library, `make eardbd.install` EAR Database Manager, `make eargmd.install` EAR Global Manager and `make commands.install` the EAR command binaries.

8.7 Installation content

This is the list of the inner installation folders and their content:

Root	Directory	Content / description
<*PREFIX*>	/lib	Libraries.
<*PREFIX*>	/lib/plugins	Plugins.
<*PREFIX*>	/bin	EAR commands.
<*PREFIX*>	/bin/tools	EAR tools for coefficients.
<*PREFIX*>	/sbin	Privileged components.
<*PREFIX*>	/man	Documentation.
<*EAR_ETC*>	/ear	Configuration file.
<*EAR_ETC*>	/ear/coeffs	Coefficient files store.
<*EAR_ETC*>	/module	EAR module.
<*EAR_ETC*>	/slurm	ear.plugstack.conf.
<*EAR_ETC*>	/systemd	EAR service files.

8.8 Fine grain tuning of EAR options

Some options such as the maximum number of CPUs or GPUs supported are defined in src/common/config files. It is not recommended to modify these files but some options and default values can be set by modifying them.

8.9 Next step

For a better overview of the installation process, return to our Quick installation guide. To continue the installation, visit the configuration page to set up properly the EAR configuration file and the SLURMs plugin stack file.

Chapter 9

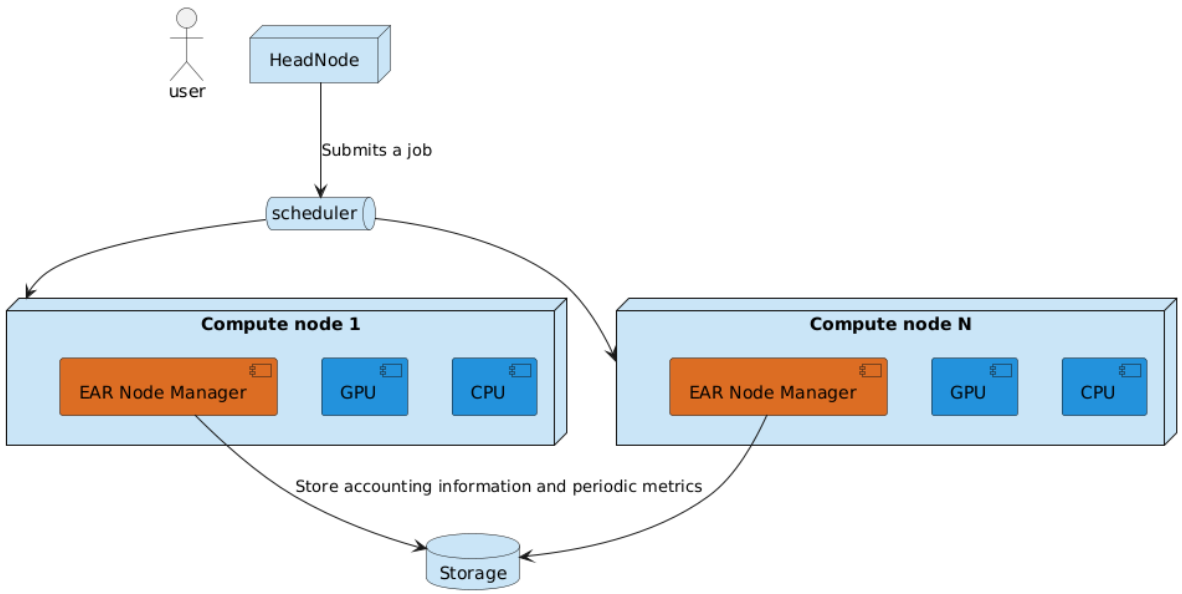
Architecture

9.1 Overview

EAR is formed by a set of components, where each of them and their relationships with each other provides a full system software which accounts the power and energy consumption of jobs and applications in a cluster, provides a runtime library for application performance monitoring and optimization which can be loaded dynamically during application execution, a global power-capping system and a flexible reporting system to fit any storage requirements for saving all the collected data, all designed to be as most transparent as possible from the user point of view. This section introduces all of these components and how they are stacked to provide different services and EAR features.

9.1.1 System power consumption and job accounting

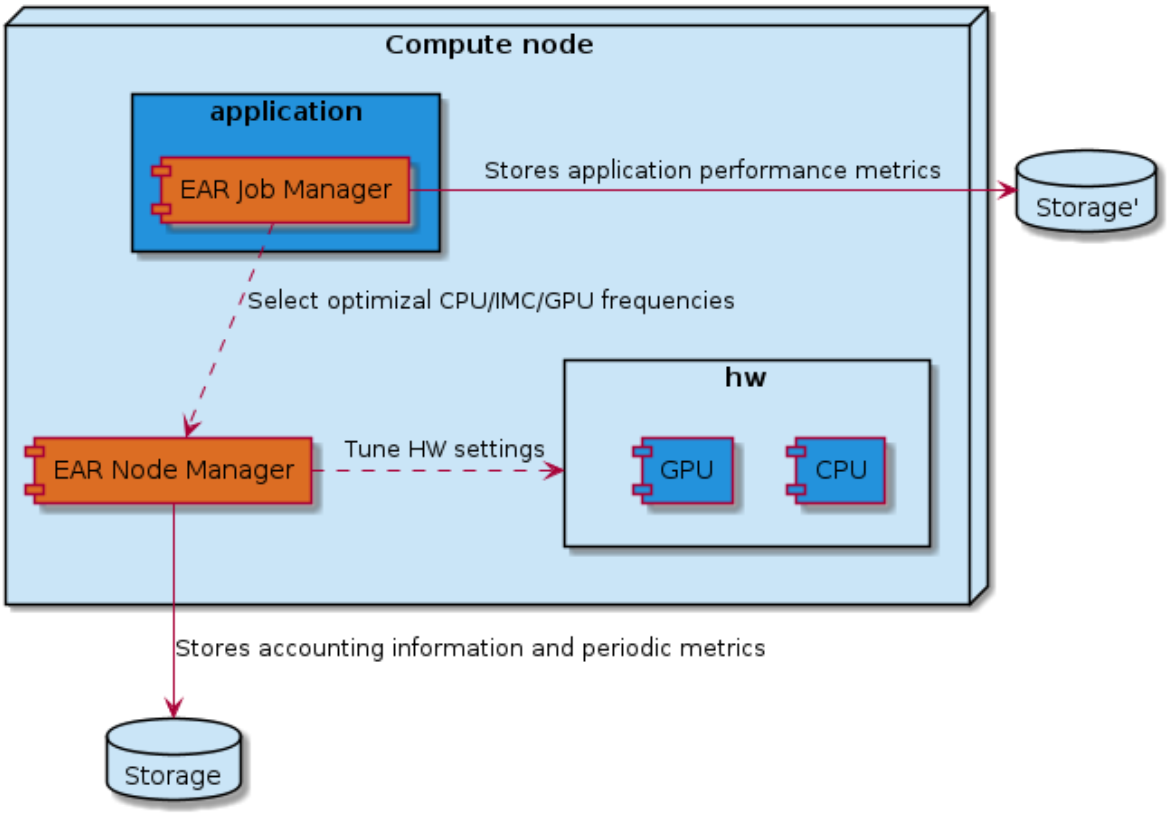
This is the most basic feature. EAR is able to collect node power consumption and report it periodically thanks to the EAR Node Manager(EARD), a Linux service which runs on each compute node. Is up to the sysadmin to decide how and where its periodic metrics are `reported`. The following figure shows this scheme.



The EAR Node Manager provides an API which can be used by a batch scheduler plug-in/hook to indicate the start/end of jobs/steps so it can account the power consumption of such entities. Currently, EAR distribution comes with a SLURM SPANK plug-in for supporting the accounting of jobs and steps in SLURM systems.

9.1.2 Application performance monitoring and energy efficiency optimization

Along with applications running in compute nodes, a runtime library can be loaded dynamically (thanks again to the batch scheduler support). The EAR Job Manager(EARL) runs within application/workflow processes, so it can collect performance metrics, which can be reported in the same way as with the Node Manager, but still configurable. Moreover, the Job Manager comes with optimization policies, which can select the optimal CPU/↩ IMC/GPU frequencies based on those performance metrics by contacting with the Node Manager. Below figure shows the interaction between these two components.



9.2 EAR Node Manager

The Node Manager (EARD) is a per-node linux service that provides privileged metrics of each node as well as a periodic power monitoring service. Said periodic power metrics can be sent to EAR's database directly, via the EAR Database Daemon (EARDBD) or by using some of the provided report plug-ins.

See the EARDBD section and the configuration page for more information about the EAR Database Manager and how to to configure the EARD to send its collected data to it.

9.2.1 Overview

EARD is the component in charge of providing any kind of services that requires privileged capabilities. Current version is conceived as an external process executed with root privileges.

It provides the following services, each one covered by one thread:

- Provides privileged metrics to EARL such as the average frequency, uncore integrated memory controller counters to compute the memory bandwidth, as well as energy metrics (DC node, DRAM and package energy).
- Implements a periodic power monitoring service. This service allows EAR package to control the total energy consumed in the system.
- Offers a remote API used by EARplug, EARGM and EAR commands. This API accepts requests such as get the system status, change policy settings or notify new job/end job events.

9.2.2 Requirements

If using the EAR Database as the storage targe, EARD connects with EARDBD service, that has to be up before starting the node daemon, otherwise values reported by EARD to be stored in the database, will be lost.

9.2.3 Configuration

The EAR Daemon uses the `$(EAR_ETC)/ear/ear.conf` file to be configured. It can be dynamically configured by reloading the service.

Please visit the EAR configuration file page for more information about the options of EARD and other components.

9.2.4 Execution

To execute this component, these `systemctl` command examples are provided:

- `sudo systemctl start eard` to start the EARD service.
- `sudo systemctl stop eard` to stop the EARD service.
- `sudo systemctl reload eard` to force reloading the configuration of the EARD service.

Log messages are generated during the execution. Use `journalctl` command to see eard message:

- `sudo journalctl -u eard -f`

9.2.5 Reconfiguration

After executing a `systemctl reload eard` command, not all the EARD options will be dynamically updated. The list of updated variables are:

```
DefaultPstates
NodeDaemonMinPstate
NodeDaemonVerbose
NodeDaemonPowermonFreq
SupportedPolicies
MinTimePerformanceAccuracy
```

To reconfigure other options such as EARD connection port, coefficients, etc., it must be stopped and restarted again. Visit the EAR configuration file page for more information about the options of EARD and other components.

9.3 EAR Database Manager

The EAR Database Daemon (EARDBD) acts as an intermediate layer between any EAR component that inserts data and the EAR's Database, in order to prevent the database server from collapsing due to getting overrun with connections and insert queries.

The Database Manager caches records generated by the EAR Library and the EARD in the system and reports it to the centralized database. It is recommended to run several EARDBDs if the cluster is big enough in order to reduce the number of inserts and connections to the database.

Also, the EARDBD accumulates data during a period of time to decrease the total insertions in the database, helping the performance of big queries. By now just the energy metrics are available to accumulate in the new metric called energy aggregation. EARDBD uses periodic power metrics sent by the EARD, the per-node daemon, including job identification details (Job Id and Step Id when executed in a SLURM system).

9.3.1 Configuration

The EAR Database Daemon uses the `$(EAR_ETC)/ear/ear.conf` file to be configured. It can be dynamically configured by reloading the service.

Please visit the `EAR configuration file page` for more information about the options of EARDBD and other components.

9.3.2 Execution

To execute this component, these `systemctl` command examples are provided:

- `sudo systemctl start eardbd` to start the EARDBD service.
- `sudo systemctl stop eardbd` to stop the EARDBD service.
- `sudo systemctl reload eardbd` to force reloading the configuration of the EARDBD service.

9.4 EAR Global Manager (System power manager)

The EAR Global Manager Daemon (EARGMD) is a cluster wide component offering cluster energy monitoring and capping. EARGM can work in two modes: manual and automatic. When running in manual mode, EARGM monitors the total energy consumption, evaluates the percentage of energy consumption over the energy limit set by the admin and reports the cluster status to the DB. When running in automatic mode, apart from evaluating the energy consumption percentage it sends the evaluation to computing nodes. EARDs passes these messages to EARL which re-applies the energy policy with the new settings.

Apart from sending messages and reporting the energy consumption to the DB, EARGM offers additional features to notify the energy consumption: automatic execution of commands is supported and mails can also automatically be sent. Both the command to be executed or the mail address can be defined in the `ear.conf`, where it can also be specified the energy limits, the monitoring period, etc.

EARGM uses periodic aggregated power metrics to efficiently compute the cluster energy consumption. Aggregated metrics are computed by EARDBD based on power metrics reported by EARD, the per-node daemon.

> **Note**: if you have multiple EARGMs running, only 1 should be used for Energy management. To turn off energy management for a certain EARGM simply set its energy value to 0.

9.4.1 Power capping

EARGM also includes an optional power capping system. Power capping can work in two different ways:

- Cluster power cap (unlimited): Each EARGM controls the power consumption of the nodes under them by ensuring the global power does not exceed a set value. While the global power is under a percentage of the global value, the nodes run without any cap. If it approaches said value, a message is sent to all nodes to set their powercap to a pre-set value (via max_powercap in the tags section of ear.conf). Should the power go back to a value under the cap, a message is sent again so the nodes run at their default value (unlimited power).
- Fine grained power cap control: Each EARGM controls the power consumption of the nodes under them and redistributes a certain budget between the nodes, allocating more to nodes who need it. It guarantees that any node has its default powercap allocation (defined by the powercap field in the tags section of ear.conf) if it is running an application.

Furthermore, when using fine grained power cap control it is possible to have multiple EARGMs, each controlling a part of the cluster, with (or without) meta-EARGMs redistributing the power allocation of each EARGM depending on the current needs of each part of the cluster. If no meta-EARGMs are specified, the power value each EARGM has will be static.

Meta-EARGMs are NOT compatible with the unlimited cluster powercap mode.

9.4.2 Configuration

The EAR Global Manager uses the `$(EAR_ETC)/ear/ear.conf` file to be configured. It can be dynamically configured by reloading the service.

Please visit the `EAR configuration file page` for more information about the options of EARGM and other components.

Additonally, 2 EARGMs can be used in the same host by declaring the environment variable EARGMID to specify which EARGM configuration each should use. If said variable is not declared, all EARGMs in the same host will read the first entry.

9.4.3 Execution

To execute this component, these `systemctl` command examples are provided:

- `sudo systemctl start eargmd` to start the EARGM service.
- `sudo systemctl stop eargmd` to stop the EARGM service.
- `sudo systemctl reload eargmd` to force reloading the configuration of the EARGM service.

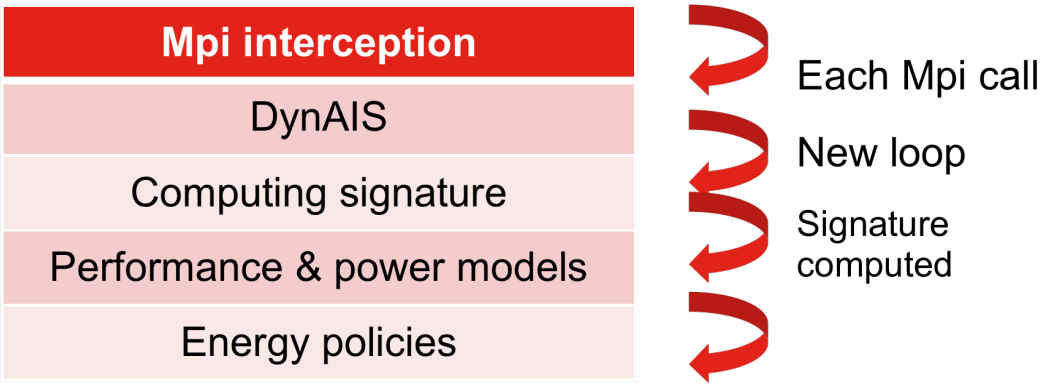
9.5 The EAR Library (Job Manager)

The EAR Library (EARL) is the core of the EAR package. The Library offers a lightweight and simple solution to select the optimal frequency for applications at runtime, with multiple power policies each with a different approach to find said frequency.

EARL uses the Daemon to read performance metrics and to send application data to EAR Database.

9.5.1 Overview

EARL is dynamically loaded next to the running applications by the EAR Loader. The Loader detects whether the application is MPI or not. In case it is MPI, it also detects whether it is Intel or OpenMPI, and it intercepts the MPI symbols through the PMPI interface, and next symbols are saved in order to provide compatibility with MPI or other profiling tools. The Library is divided in several stages summarized in the following picture:



1. Automatic **detection** of application outer loops. This is done by intercepting MPI calls and invoking the Dynamic Application Iterative Structure detector algorithm. **DynAIS** is highly optimized for new Intel architectures, reporting low overhead. For non-MPI applications, EAR implements a time-guided approach.
2. Computation of the **application signature**. Once DynAIS starts reporting iterations for the outer loop, EAR starts to compute the application signature. This signature includes: iteration time, DC power consumption, bandwidth, cycles, instructions, etc. Since the DC power measurements error highly depends on the hardware, EAR automatically detects the hardware characteristics and sets a minimum time to compute the signature in order to minimize the average error.

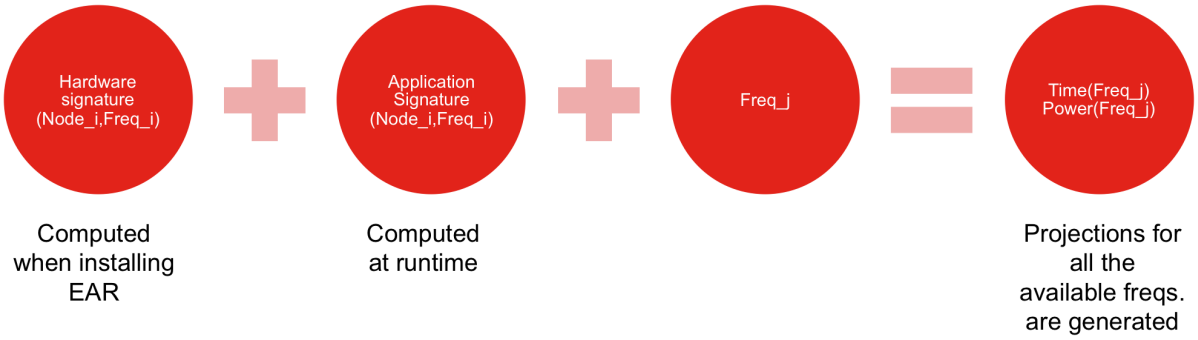
$$Power(fn) = A(Rf,fn)*Power(Rf) + B(Rf,fn)*TPI(Rf) + C(Rf,fn)$$

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

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

The loop signature is used to **classify the application activity** in different phases. The current EAR version supports the following phases for: IO bound, CPU computation and GPU idle, CPU busy waiting and GPU computing, CPU-GPU computation, and CPU computation (for CPU only nodes). For phases including CPU computation, the optimization policy is applied. For other phases, the EAR library implements some predefined CPU/Memory/GPU frequency settings.

1. **Power and performance projection**. EAR has its own performance and power models which requires the application and the system signatures as an input. The system signature is a set of coefficients characterizing each node in the system. They are computed during the learning phase at the EAR configuration step. EAR projects the power used and computing time (performance) of the running application for all the available frequencies in the system. These models are applied to CPU metrics and projects CPU performance and power when varying the CPU frequency. Using these projections the optimization policy can select the optimal CPU memory.



1. **Apply** the selected energy optimization policy. EAR includes two power policies to be selected at runtime: *minimize time to solution* and *minimize energy to solution*, if permitted by the system administrator. At this point, EAR executes the power policy, using the projections computed in the previous phase, and selects the optimal frequency for an application and its particular run. An additional policy, *monitoring only* can also be used, but in this case no changes to the running frequency will be made but only the computation and storage of the application signature and metrics will be done. The short version of the names is used when submitting jobs (min_energy, min_time, monitoring). Current policies already includes memory frequency selection but in this case it is not based on models, it is a guided search. Check in your installation in the memory frequency optimization is enabled by default. In case the application is MPI, the policies already classifies the processes as balanced or unbalanced. In case they are unbalanced, a per-process CPU frequency is applied.

Some specific configurations are modified when jobs are executed sharing nodes with other jobs. For example the memory frequency optiization is disabled. See section environment variables page for more information on how to tune the EAR library optimization using environment variables.

9.5.2 Configuration

The Library uses the `$(EAR_ETC)/ear.conf` file to be configured. Please visit the `EAR configuration file page` for more information about the options of EARL and other components.

EARL receives its specific settings through a shared memory regions initialized by EARD.

9.5.3 Usage

For information on how to run applications alongside with EARL read the User guide. Next section contains more information regarding EAR's optimisation policies.

9.5.4 Classification

In the EARL's pipeline, classification is a step that optimizes the power and performance projections of the energy models. The idea behind it is that, upon identifying the type of activity of applications, we can adapt its execution according to the architecture's exploited resources.

As explained in the EAR Library section, EAR accounts for different execution phases, which can be separated into **CPU computation** and **non-CPU computation**. Bear in mind that, for now, CPU computation phases only take into account the activity of the CPU, and not that of the GPU. Now, since the optimization policy is applied for these execution phases, and it depends on the projections made by the energy models, the classification optimizes these projections by indicating if they have to try increasing or decreasing pstates.

In EAR, CPU computation phases include

- **CPU-bound** phases, which are characterized by intensive usage of the CPU for calculus-related operations (normally measured through the GFLOPS and CPI)
- **MEMORY-bound** phases, characterized by the intensity of calls to (main) memory (normally measured through MEM_GBS and TPI)
- **MIX** phases, which are in-between these two

Given the complexity of correctly making distinctions between these execution phases, the classification becomes a step that requires an adequate strategy to tackle it. Thus, in this section we will present the strategies proposed & implemented in the EARL.

9.5.4.1 Default model

EAR's default classification model is based on the application of a fixed set of ranges to CPI and MEM_GBS metrics. These ranges, defined according to the architecture's characteristics via expert knowledge, would allow identifying the different execution phases on a fundamental level.

Input

This strategy, available since EAR's installation, takes 4 thresholds, 2 for delimiting the CPI and memory bandwidth (i.e., MEM_GBS) of CPU-bound applications, and 2 more for delimiting those of MEMORY-bound ones. For instance, the values proposed for *Sapphire Rapids* type of node are

- CPI of CPU-bound apps: 0.4
- MEM_GBS of CPU-bound apps: 180
- CPI of MEMORY-bound apps: 0.4
- MEM_GBS of MEMORY-bound apps: 250

Classification philosophy

With these thresholds defined, the classification proceeds as follows

```
Let S be the last registered signature
Let CPU be the struct with the CPU-bound-related thresholds
Let MEM be the struct with the MEMORY-bound-related thresholds
IF (S->CPI <= CPU->CPI && S->MEM_GBS <= CPU->MEM_GBS)
    Mark app as CPU-bound
ELSE IF (S->CPI >= MEM->CPI && S->MEM_GBS >= MEM->MEM_GBS)
    Mark app as MEMORY-bound
ELSE
    Mark app as MIX
```

Let us go over the cases considered by the strategy:

1. To begin with, the model checks if the application is CPU-bound by checking if the CPI and MEM_GBS of the application are *below* the CPU thresholds. The sign of this comparison is due to the fact that we expect a CPU-bound application to be executing lots of instructions (thus having a small CPI) and not bounded by memory (thus registering a low memory bandwidth).
2. If the strategy finds that the app is not CPU-bound, it checks whether the considered metrics are *above* the MEM thresholds. The sign of this comparison is due to the fact that we expect a MEMORY-bound app to be using a considerable amount of memory bandwidth (thus having a big MEM_GBS) and not executing too many instructions (thus having a big CPI aswell).
3. If none of these conditions are met, we label the app as MIX.

The strength of this approach is its quickness and effectiveness, given that the classification is based upon the performance typically expected from the execution phases considered.

9.5.4.2 Roofline model

The roofline model combines floating point performance, memory traffic and operational intensity to characterize the activity of an application based on the performance limitations of the hardware [2]. For EAR, it allows for identifying execution phase types in a simple & quick way in runtime.

Input

To use this strategy in EAR, we need the floating point performance and memory traffic peaks, which can be computed either theoretically or empirically.

Theoretical roofline

To get the theoretical peaks, we propose the usage of equations (1) (2) as follows:

$$\text{Peak flops} = f_{\text{CPU}} \times \#\text{cores} \times \#\text{sockets} \times \#\frac{\text{FLOPS}}{\text{cycle}} \times \#\text{FMA units}$$

$$\text{Peak memory bandwidth} = f_{\text{memory}} \times \#\frac{\text{bytes}}{\text{cycle}} \times \#\text{memory channels} \times \#\text{sockets}$$

NOTE: it is usually assumed that.
```
- num. of FMA units= 2
- bytes/cycle=8
- flops/cycle=16
```

To give an idea of how to apply these equations, let us give an example on how we would apply them on EPYC 9654 nodes:

$$\text{Peak flops} = 3.7\ \text{GHz} \times 96\ \text{cores} \times 2\ \text{sockets} \times 16\frac{\text{FLOPS}}{\text{cycle}} \times 2\ \text{FMA units} \quad = 22.732,8\ \text{GFlops}$$

$$\text{Peak memory bandwidth} = 4.8\ \text{GHz} \times 8\frac{\text{bytes}}{\text{cycle}} \times 12\ \text{memory channels} \times 2\ \text{sockets} \quad = 921,6\ \text{GB/s}$$

Empirical roofline

To get the empirical peaks, it is suggested to use the STREAM benchmark [1] and the HPL benchmark [5] for harvesting the memory bandwidth floating point performance peaks, respectively.

Once the peaks have been properly computed, we store them in a file of the form `roofline.{tag}.data`, where `tag` corresponds to the tag of the node partition, which will depend on the environment the user is working on (check Tags for more information). The format to store the peaks is:
```
{peak memory bandwidth} {peak floating performance}
```

In summary, it is only required a plain text file, with '`roofline.{architecture}.data` as its filename, and only containing both peaks ordered. Moreover, this file is expected to be stored in the `$EAR_ETC/ear/coeffs` directory; check the EARL configuration section for more details on this.

Following the EPYC 9654 example, we would create a file called `roofline.epyc9654.data` containing:
```
921.6 22732.8
```

Classification strategy

Once EAR has access to the roofline peaks, the classification proceeds as follows:
```
Let S be the last registered signature
Let PEAK be the struct with the peaks of the architecture
IF (S->GFLOPS / S->MEM_GBS >= PEAK->GFLOPS / PEAK->MEM_GBS)
    Mark app as CPU-bound
ELSE IF (S->MEM_GBS >= PEAK->MEM_GBS * 0.75)
    Mark app as MEMORY-bound
ELSE
    Mark app as MIX
```

Let us go over it step by step:

1. We begin by checking if the app is *CPU-bound*, which is equivalent to checking if the operational intensity is bigger than the threshold defined by the peaks of the architecture.
2. If the app is not *CPU-bound*, we check if the app has a memory bandwidth close enough (in absolute units) to the peak of the architecture. To do so, we measure it by defining a "similarity threshold" (in our example, 0.75). If it is close enough, the app is labelled as *MEMORY-bound*.
3. If the app is neither of them, then it is labelled as *MIX*.

This way, we adapt the original roofline model to include the *MIX* execution phase while maintaining its classification philosophy.

9.5.4.3 K-medoids

K-medoids is a clustering method based in the k-means [6], but instead of using *centroids* as representatives of the classes, it uses *medoids* (i.e., elements of the dataset) [7]. This strategy's usage is encouraged once EAR has access to enough user data to train such a model. Also, it is flexible enough to be regenerated over time, thus accounting for changes in the type of applications executed by users as well as becoming more robust over time.

Furthermore, to achieve a balance between classification quality and runtime performance, the model used by EAR defines medoids as subsets of signature metrics conformed by CPI, TPI, GFLOPS and MEM_GBS. This way, it combines metrics both from the architecture and the application's computational activity.

Finally, it is of interest to note that, to ensure that the classification is properly conducted, the data is previously standardized [8].

Input

To use this strategy, two plain text files are needed: one for the standardization, and another one for the medoids. Regarding the first one, it stores the mean and standard deviations in the following format:

```
{CPI std} {CPI mean} {TPI std} {TPI mean} {GFLOPS std} {GFLOPS mean} {MEM_GBS std} {MEM_GBS mean}
```

This first file must follow the format name `extremes.{tag}.data`, where `tag` corresponds to the tag of the node partition.

Now, regarding the second file, it stores the medoids in the following format:

```
{CPU-bound CPI} {CPU-bound TPI} {CPU-bound GFLOPS} {CPU-bound MEM_GBS} {MEMORY-bound CPI} {MEMORY-bound TPI}
      {MEMORY-bound GFLOPS} {MEMORY-bound MEM_GBS} {MIX CPI} {MIX TPI} {MIX GFLOPS} {MIX MEM_GBS}
```

This file must also follow a format name `medoids.{tag}.data`, where `tag` corresponds to the tag of the node partition.

NOTE: the `tag` will depend on the environment the user is working on. Check Tags for more information on this.

NOTE 2: as in the *roofline* case, it is expected that both `medoids` and `extremes` files are stored in the `$EAR_ETC/ear/coeffs` directory. Check the EARL configuration section for more details on this.

Classification strategy

The classification proceeds as follows

```
Let S be the last signature registered by EAR
Let V := [S->CPI, S->TPI, S->GFLOPS, S->MEM_GBS]
Standardize vector V
Check Euclidean distance from V to the medoids and label signature accordingly
```

Let us go over the steps followed by this pseudocode:

1. For each loop signature registered, we standardize its CPI, TPI, GFLOPS and MEM_GBS. To do so, EAR preloads the extremes (i.e., mu and sigma of each metric) at the beginning of the execution.
2. Once processed, we check the Euclidean distance of this vector to the different medoids, which are also preloaded (just like extremes).
3. With these distances computed, we identify the current execution phase of the app by checking which one is closer to the signature.

With these steps, EAR's classification becomes more flexible and equally robust.

As a final note, the library chooses the classification strategy in the following way:

```
IF medoids are available
    Load K-medoids model
ELSE IF roofline is available
    Load roofline model
ELSE
    Load default model
```

Thus, the user must be aware not only of the availability of each strategy, but also how the library prioritizes some models over others.

9.5.4.4 References

[1] McCalpin, John. "STREAM: Sustainable memory bandwidth in high performance computers." *http://www.cs.virginia.edu/stream/* (2006).

[2] Williams, S., Waterman, A., & Patterson, D. (2009). Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, *52*(4), 65-76.

[3] Andreolli, C., Thierry, P., Borges, L., Skinner, G., Yount, C., Jeffers, J., & Reinders, J. (2015). Characterization and optimization methodology applied to stencil computations. *High Performance Parallelism Pearls*, 377-396.

[4] Bakos, J. D. (2016). Multicore and data-level optimization. In Embedded Systems (pp. 49–103). Elsevier. https://doi.org/10.1016/b978-0-12-800342-8.00002-x

[5] Petitet, Antoine. "HPL-a portable implementation of the high-performance Linpack benchmark for distributed-memory computers." *http://www.netlib.org/benchmark/hpl/* (2004).

[6] Blömer, J., Lammersen, C., Schmidt, M., & Sohler, C. (2016). Theoretical analysis of the k-means algorithm–a survey. *Algorithm Engineering: Selected Results and Surveys*, 81-116.

[7] Park, H. S., & Jun, C. H. (2009). A simple and fast algorithm for K-medoids clustering. *Expert systems with applications*, *36*(2), 3336-3341.

[8] Gal, M. S., & Rubinfeld, D. L. (2019). Data standardization. *NYUL Rev.*, *94*, 737.

9.5.5 Policies

EAR offers three energy policies plugins: `min_energy`, `min_time` and `monitoring`. The last one is not a power policy, is used just for application monitoring where CPU frequency is not modified (neither memory or GPU frequency). For application analysis `monitoring`can be used with specific CPU, memory and/or GPU frequencies.

The energy policy is selected by setting the `--ear-policy=policy` option when submitting a SLURM job. A policy parameter, which is a particular value or threshold depending on the policy, can be set using the flag `--ear-policy-th=value`. Its default value is defined in the configuration file, for more information check the configuration page for more information.

9.5.5.1 `min_energy`

The goal of this policy is to minimise the energy consumed with a limit to the performance degradation. This limit is is set in the SLURM `--ear-policy-th` option or the configuration file. The `min_energy` policy will select the optimal frequency that minimizes energy enforcing (performance degradation <= parameter). When executing with this policy, applications starts at default frequency(specified at ear.conf).

```
PerfDegr = (CurrTime - PrevTime) / (PrevTime)
```

9.5.5.2 min_time

The goal of this policy is to improve the execution time while guaranteeing a minimum ratio between performance benefit and frequency increment that justifies the increased energy consumption from this frequency increment. The policy uses the SLURM parameter option mentioned above as a minimum efficiency threshold.

Example: if `--ear-policy-th=0.75`, EAR will prevent scaling to upper frequencies if the ratio between performance gain and frequency gain do not improve at least 75% (PerfGain >= (FreqGain * threshold).

```
PerfGain=(PrevTime-CurrTime)/PrevTime
FreqGain=(CurFreq-PrevFreq)/PrevFreq
```

When launched with `min_time` policy, applications start at a default frequency (defined at `ear.conf`). Check the configuration page for more information.

Example: given a system with a nominal frequency of 2.3GHz and default P_STATE set to 3, an application executed with `min_time` will start with frequency `F\\\[i\\\]=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 the Figure 1. If performance gain is greater or equal than threshold, the policy will check with the next performance projection `F\\\[i+2\\\]`. If the performance gain computed is less than threshold, the policy will select the last frequency where the performance gain was enough, preventing the waste of energy.

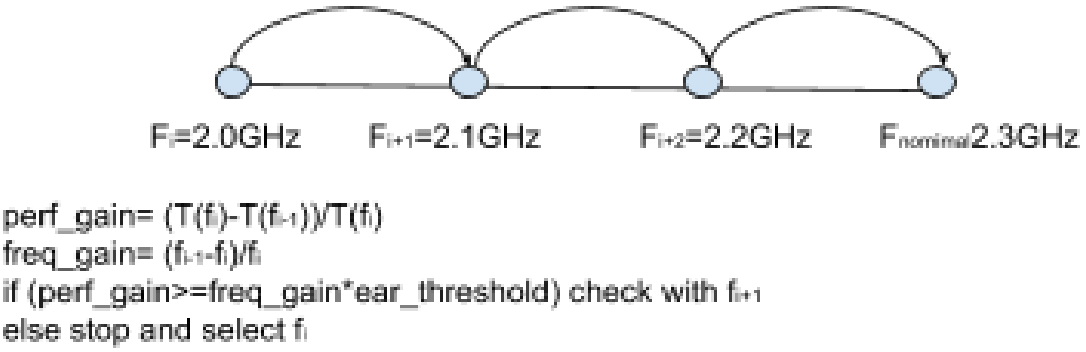


Figure 1: `min_time` uses the threshold value as the minimum value for the performance gain between `F\\\[i\\\]` and `F\\\[i+1\\\]`.

9.6 EAR Loader

The EAR Loader is the responsible for loading the EAR Library. It is a small and lightweight library loaded by the EAR SLURM Plugin (through the `LD_PRELOAD` environment variable) that identifies the user application and loads its corresponding EAR Library distribution.

The Loader detects the underlying application, identifying the MPI version (if used) and other minor details. With this information, the loader opens the suitable EAR Library version.

As can be read in the EARL page, depending on the MPI vendor the MPI types can be different, preventing any compatibility between distributions. For example, if the MPI distribution is OpenMPI, the EAR Loader will load the EAR Library compiled with the OpenMPI includes.

You can read the installation guide for more information about compiling and installing different EARL versions.

9.7 EAR SLURM plugin

EAR SLURM plug-in allows to dynamically load and configure the EAR Library for the SLURM jobs (and steps), if the flag `--ear=on` is set or if it is enabled by default. Additionally, it reports any jobs that start or end to the nodes' EARDs for accounting and monitoring purposes.

9.7.1 Configuration

Visit the SLURM SPANK plugin section on the configuration page to set up properly the SLURM `/etc/slurm/plugstack.conf` file.

You can find the complete list of EAR SLURM plugin accpeted parameters in the user guide.

9.8 EAR Data Center Monitor

It is a new EAR service for Data Center monitoring. In particular, it targets elements different than computational nodes which are already monitored by the EARD running in compute nodes. It has a [dedicated section](EDCMON) you can read for more information.

9.9 EAR application API

EAR offers a user API for applications. The current EAR version only offers two sets of functions:

- To measure the energy consumption
- To set the cpu and gpu frequencies .
- `int ear_connect()`
- `int ear_energy(unsigned long \\\*energy_mj, unsigned long \\\*time_ms)`
- `void ear_energy_diff(unsigned long ebegin, unsigned long eend, unsigned long \\\*ediff, unsigned long tbegin, unsigned long tend, unsigned long \\\*tdiff)`
- `int ear_set_cpufreq(cpu_set_t \\\*mask,unsigned long cpufreq);`
- `int ear_set_gpufreq(int gpu_id,unsigned long gpufreq)`
- `int ear_set_gpufreq_list(int num_gpus,unsigned long \\\*gpufreqlist)`
- `void ear_disconnect()`

EAR's header file and library can be found at $EAR_INSTALL_PATH/include/ear.h and $EAR_INSTALL_↩ PATH/lib/libEAR_api.so respectively. The following example reports the energy, time, and average power during that time for a simple loop including a `sleep(5)`.

```
#define _GNU_SOURCE
#include <ear.h>

int main(int argc,char *argv[])
{
  unsigned long e_mj=0,t_ms=0,e_mj_init,t_ms_init,e_mj_end,t_ms_end=0;
  unsigned long ej,emj,ts,tms,os,oms;
  unsigned long ej_e,emj_e,ts_e,tms_e,os_e,oms_e;
  int i=0;
```

```
  struct tm *tstamp,*tstamp2,*tstamp3,*tstamp4;
  char s[128],s2[128],s3[128],s4[128];

  /* Connecting with ear */
  if (ear_connect()!=EAR_SUCCESS)
  {
    printf("error connecting eard\n");
    exit(1);
  }

  /* Reading energy */
  if (ear_energy(&e_mj_init,&t_ms_init)!=EAR_SUCCESS)
  {
    printf("Error in ear_energy\n");
  }
  while(i<5)
  {
    sleep(5);

    /* READING ENERGY */
    if (ear_energy(&e_mj_end,&t_ms_end)!=EAR_SUCCESS)
    {
      printf("Error in ear_energy\n");
    }
    else
    {
      ts=t_ms_init/1000;
      ts_e=t_ms_end/1000;
      tstamp=localtime((time_t *)&ts);
      strftime(s, sizeof(s), "%c", tstamp);
              tstamp2=localtime((time_t *)&ts_e);
              strftime(s2, sizeof(s), "%c", tstamp2);
      printf("Start time %s End time %s\n",s,s2);
      ear_energy_diff(e_mj_init,e_mj_end, &e_mj, t_ms_init,t_ms_end,&t_ms);
      printf("Time consumed %lu (ms), energy consumed %lu(mJ),
             Avg power %lf(W)\n",t_ms,e_mj,(double)e_mj/(double)t_ms);
      e_mj_init=e_mj_end;
      t_ms_init=t_ms_end;
    }
    i++;
  }
  ear_disconnect();
}
```

Chapter 10

High availability support

EAR is designed to provide support for those systems where High Availability (HA) of services/data is a must. Currently, just the EAR Database Manager and the EAR Database components offer HA support. The latter can be replaced by industry software like `Galera Cluster`.

10.1 EAR Database Manager HA

Database daemons can be configured and deployed in pairs following a *server*/∗mirror∗ schema. Buffering before reporting to the Database is the main duty of this Daemon, so having a mirror ensures no data lose in the case the service stops or the node where it is being running fails unexpectedly. Both services buffer data, but just the server inserts it to the Database. If it fails, the mirror starts insertion process.

You can read how to configure two EAR Database Managers on the Configuration Guide's Island description section.

10.2 EAR Database HA

If the EAR Database is not deplyed on a HA system, you can manually configure EAR to report data on two databases. Just three actions must be performed:

- Create two databases on two different servers. You have two ways:
  1. Use *edb_create* command.
  2. Use `edb_create -o` and redirect the output to a file. You will have the SQL query set saved to be executed directly in the DB server (mysql). ∗
- Specify **DBSECIP** field in the `ear.conf` (and propagate changes).
- Restart Database Manager daemons.
- Only supported in mysql.
- You must add a semi-colon (;) at the end of each line.

Chapter 11

EAR configuration

11.1 EAR Configuration requirements

The following requirements must be met for EAR to work properly:

11.1.1 EAR paths

EAR folders EAR uses two paths for EAR configuration:

- **EAR_TMP:** *tmp_ear_path* must be a private folder per compute node. It must have read/write permissions for normal users. Communication files are created here. It must be created by the admin. For instance: `mkdir /var/ear; chmod ugo +rwx /var/ear`.
- **EAR_ETC:** *etc_ear_path* can be installed in shared folder (e.g., GPFS) or can be replicated because it has very few data and it is modified at a very low frequency (**ear.conf** and coefficients). You can prevent other users to read this file since it contains Database client's passwords. Coefficients can be installed in a different path specified at `configure` time with `COEFFS` flag. Both `ear.conf` and coefficients must be readable in all the nodes (login, compute and service nodes).

ear.conf `ear.conf` is an ascii file setting default values and cluster descriptions. An `ear.conf` is automatically generated based on a **ear.conf.in** template. However, the administrator must include installation details such as hostname details for EAR services, ports, default values, and the list of nodes. For more details, check EAR configuration file below.

11.1.2 DB creation and DB server

MySQL or PostgreSQL database: EAR saves data in a MySQL/PostgreSQL DB server. EAR DB can be created using `edb_create` command provided (MySQL/PostgreSQL server must be running and root access to the DB is needed).

11.1.3 EAR SLURM plug-in

EAR SLURM plug-in can be enabled by adding an additional line at the `/etc/slurm/plugstack.conf` file. You can copy from the `ear_etc_path/slurm/ear.plugstack.conf` file).

Another way to enable it is to create the directory `/etc/slurm/plugstack.conf.d` and copy there the `ear_etc_path/slurm/ear.plugstack.conf` file. On that case, the content of `/etc/slurm/plugstack.conf` must be `include /etc/slurm/plugstack.conf.d/\\\*`.

11.2 EAR configuration file

The **ear.conf** is a text file describing the EAR package behaviour in the cluster. It must be readable by all compute nodes and by nodes where commands are executed. Two `ear.conf` templates are generated with default values and will be installed as reference when executing `make etc.install`.

Usually the first word in the configuration file expresses the component related with the option. Lines starting with # are comments. A test for `ear.conf` file can be found in the path `src/test/functionals/ear_conf`. It is recommended to test it since the `ear.conf` parser is very sensible to errors in the `ear.conf` syntax, spaces, newlines, etc.

In order to improve the readability of the ear.con, EAR version 5.0 includes a new clause "include" that case be used to include additional files with parts of the configurations such as tags or the list of nodes. The syntax is:

```
include=absolute_path
```

11.2.1 Database configuration

```
# The IP of the node where the MariaDB (MySQL) or PostgreSQL server process is running. Current version uses
      same names for both DB servers.
DBIp=172.30.2.101
# Uncomment and add a secondary IP for high availability.
# If specified, the mysql plugin will submit data to a second DB automatically .
# Not supportd with other report plugins.
# DBSECIP=add_secondary_ip_for_ha

# Port in which the server accepts the connections.
DBPort=3306

# MariaDB user that services will use. Needs INSERT/SELECT privileges. Used by the EARDBD.
DBUser=eardbd_user
# Password for the previous user. If left blank or commented it will assume the user has no password.
DBPassw=eardbd_pass
# Database user that the commands (eacct, ereport) will use. Only uses SELECT privileges.
DBCommandsUser=ear_commands
# Password for the previous user. If left blank or commented it will assume the user has no password.
DBCommandsPassw=commandspass

# Name of EAR's database in the server.
DBDatabase=EAR

# Maximum number of connections of the commands user to prevent server
# saturation/malicious actuation. Applies to DBCommandsUser.
DBMaxConnections=20
# The following specify the granularity of data reported to database.
# Extended node information reported to database (added: temperature, avg_freq, DRAM and PCK energy in power
      monitoring).
DBReportNodeDetail=1
# Extended signature hardware counters reported to database.
DBReportSigDetail=1
# Set to 1 if you want Loop signatures to be reported to database.
DBReportLoops=1
```

11.2.2 EARD configuration

```
# The port where the EARD will be listening.
NodeDaemonPort=50001

# Frequency used by power monitoring service, in seconds.
NodeDaemonPowermonFreq=60
# Maximum supported frequency (1 means nominal, no turbo).
NodeDaemonMinPstate=1
# Enable (1) or disable (0) the turbo frequency.
NodeDaemonTurbo=0

# Enables the use of the database.
NodeUseDB=1
# Inserts data to MySQL by sending that data to the EARDBD (1) or directly (0).
NodeUseEARDBD=1
# '1' means EAR is controlling frequencies at all times (targeted to production systems) and 0 means EAR
      will not change the frequencies when users are not using EAR library (targeted to benchmarking
      systems).
NodeDaemonForceFrequencies=1

# The verbosity level [0..4]
```

```
NodeDaemonVerbose=1
# When set to 1, the output is saved at '$EAR_TMP'/eard.log (common configuration) as a log file. Otherwsie,
      stderr is used.
NodeUseLog=1

# Report plug-ins to be used by the EARD. Default= eardbd.so.
# Add extra plug-ins by separating with colons (e.g., eardbd.so:plugin1.so).
EARDReportPlugins=eardbd.so
```

11.2.3 EARDBD configuration

```
# Port where the EARDBD server is listening.
DBDaemonPortTCP=50002
# Port where the EARDBD mirror is listening.
DBDaemonPortSecTCP=50003
# Port used to synchronize the server and mirror.
DBDaemonSyncPort=50004

# In seconds, interval of time of accumulating data to generate an energy aggregation.
DBDaemonAggregationTime=60
# In seconds, time between inserts of the buffered data.
DBDaemonInsertionTime=30
# Memory allocated per process. These allocations are used for buffering the data
# sent to the database by EARD or other components. If there is a server and a
# mirror in a node a double of that value will be allocated. It is expressed in MegaBytes.
DBDaemonMemorySize=120

# When set to 1, EARDBD uses a '$EAR_TMP'/eardbd.log file as a log file.
DBDaemonUseLog=1

# Report plug-ins to be used by the EARDBD. Default= mysql.so.
# Add extra plug-ins by separating with colons (e.g., mysql.so:plugin1.so).
EARDBDReportPlugins=mysql.so
```

11.2.4 EARL configuration

```
# Path where coefficients are installed, usually $EAR_ETC/ear/coeffs.
CoefficientsDir=/path/to/coeffs

# NOTE: It is not recommended to change the following
# attributes if you are not an expert user.
# Number of levels used by DynAIS algorithm.
DynAISLevels=10
# Windows size used by DynAIS, the higher the size the higher the overhead.
DynAISWindowSize=200
# Maximum time (in seconds) that EAR will wait until a signature is computed. After this value, if no
      signature is computed, EAR will go to periodic mode.
DynaisTimeout=15
# Time in seconds to compute every application signature when the EAR goes to periodic mode.
LibraryPeriod=10
# Number of MPI calls whether EAR must go to periodic mode or not.
CheckEARModeEvery=1000
# EARL default report plug-ins
EARLReportPlug-ins=eard.so
```

11.2.5 EARGM configuration

You can skip this section if EARGM is not used in your installation.

```
# Verbosity
EARGMVerbose=1
# When set to 1, the output is saved in 'TmpDir'/eargmd.log (common configuration) as a log file.
EARGMUseLog=1
EARGMPort=50000
# Email address to report the warning level (and the action taken in automatic mode).
EARGMMail=nomail
# Period T1 and T2 are specified in seconds (ex. T1 must be less than T2, ex. 10min and 1 month).
EARGMEnergyPeriodT1=90
EARGMEnergyPeriodT2=259200
# '-' are Joules, 'K' KiloJoules and 'M' MegaJoules.
EARGMEnergyUnits=K
# Energy limit applies to EARGMPeriodT2.
EARGMEnergyLimit=550000
# Use aggregated periodic metrics or periodic power metrics.
# Aggregated metrics are only available when EARDBD is running.
EARGMEnergyUseAggregated=1
# Two modes are supported '0=manual' and '1=automatic'.
```

```
EARGMEnergyMode=0
# Percentage of accumulated energy to start the warning DEFCON level L4, L3 and L2.
EARGMEnergyWarningsPerc=85,90,95
# T1 "grace" periods between DEFCON before re-evaluate.
EARGMEnergyGracePeriods=3
# Format for action is: command_name energy_T1 energy_T2  energy_limit T2 T1  units "
# This action is automatically executed at each warning level (only once per grace periods).
EARGMEnergyAction=no_action

# Period at which the powercap thread is activated.
EARGMPowerPeriod=120
# Powercap mode: 0 is monitoring, 1 is hard powercap, 2 is soft powercap.
EARGMPowerCapMode=1
# Admins can specify to automatically execute a command in
# EARGMPowerCapSuspendAction when total_power >= EARGMPowerLimit*EARGMPowerCapResumeLimit/100
EARGMPowerCapSuspendLimit=90
# Format for action is: command_name current_power current_limit total_idle_nodes total_idle_power
EARGMPowerCapSuspendAction=no_action
# Admins can specify to automatically execute a command in EARGMPowerCapResumeAction
# to undo EARGMPowerCapSuspendAction when total_power >= EARGMPowerLimit*EARGMPowerCapResumeLimit/100.
# Note that this will only be executed if a suspend action was executed previously.
EARGMPowerCapResumeLimit=40
# Format for action is: command_name current_power current_limit total_idle_nodes total_idle_power
EARGMPowerCapResumeAction=no_action
# Sets the report plugins to use for EARGM warning and events accounting
EARGMReportPlugins=mysql.so


# EARGMs must be specified with a unique id, their node and the port that receives
# remote connections. An EARGM can also act as meta-eargm if the meta field is filled,
# and it will control the EARGMs whose ids are in said field. If two EARGMs are in the
# same node, setting the EARGMID environment variable overrides the node field and
# chooses the characteristics of the EARGM with the correspoding id.

# Only one EARGM can currently control the energy caps, so setting the rest to 0 is recommended.
# energy = 0  -> energy_cap disabled
# power  = 0  -> powercap disabled
# power  = N  -> powercap budget for that EARGM (and the nodes it controls) is N
# power  = -1 -> powercap budget is calculated by adding up the powercap set to each of the nodes under its
      control.
#                This is incompatible with nodes that have their powercap unlimited (powercap = 1)
EARGMId=1 energy=1800 power=600 node=node1 port=50100 meta=1,2,3
EARGMId=2 energy=0 power=500 node=node1 port=50101
EARGMId=3 energy=0 power=500 node=node2 port=50100
```

11.2.6 Common configuration

```
# Default verbose level
Verbose=0
# Path used for communication files, shared memory, etc. It must be PRIVATE per
# compute node and with read/write permissions. $EAR_TMP
TmpDir=/tmp/ear
# Path where coefficients and configuration are stored. It must be readable in all compute nodes. $EAR_ETC
EtcDir=/path/to/etc
InstDir=/path/to/inst

# Network extension: To be used in case the DC has more than one
# network and a special extension needs to be used for global commands
#NetworkExtension=
```

11.2.7 EAR Authorized users/groups/accounts

Authorized users that are allowed to change policies, thresholds and frequencies are supposed to be administrators. A list of users, Linux groups, and/or SLURM accounts can be provided to allow normal users to perform that actions. Only normal Authorized users can execute the learning phase.

```
AuthorizedUsers=user1,user2
AuthorizedAccounts=acc1,acc2,acc3
AuthorizedGroups=xx,yy
```

11.2.8 Energy tags

Energy tags are pre-defined configurations for some applications (EAR Library is not loaded). This energy tags accept a user ids, groups and SLURM accounts of users allowed to use that tag.

```
# General energy tag
EnergyTag=cpu-intensive pstate=1
# Energy tag with limited users
EnergyTag=memory-intensive pstate=4 users=user1,user2 groups=group1,group2 accounts=acc1,acc2
```

11.2.9 Tags

Tags are used for architectural descriptions. Max. AVX frequencies are used in predictor models and are SKU-specific. At least a default tag is mandatory to be included for a cluster to properly work.

The **min_power**, **max_power** and **error_power** are threshold values that determine if the metrics read might be invalid, and a warning message to syslog will be reported if the values are outside of said thresholds. The **error_power** field is a more extreme value that if a metric surpasses it, said metric will not be reported to the DataBase.

A special energy plug-in or energy model can be specified in a tag that will override the global values previously defined in all nodes that have this tag associated with them.

Powercap set to 0 means powercap is disabled and cannot be enabled at runtime. Powercap set to 1 means no limits on power consumption but a powercap can be set without stopping eard. List of accepted options:

- max_avx512 (GHz)
- max_avx2 (GHz)
- max_power (W)
- min_power (W)
- error_power (W)
- coeffs (filename)
- powercap (W)
- powercap_plugin (filename)
- energy_plugin (filename)
- gpu_powercap_plugin (filename)
- max_powercap (W)
- gpu_def_freq (KHz)
- cpu_max_pstate (0..max_pstate)
- imc_max_pstate (0..max_imc_pstate)
- energy_model (filename)
- imc_max_freq (GHz)
- imc_min_freq (GHz)
- idle_governor (governor name)
- idle_pstate (0..max_pstate)

```
Tag=6148 default=yes max_avx512=2.2 max_avx2=2.6 max_power=500 powercap=1 max_powercap=600 gpu_def_freq=1.4
      energy_model=avx512_model.so energy_plugin=energy_nm.so powercap_plugin=dvfs.so
      gpu_powercap_plugin=gpu.so min_power=50 error_power=600 coeffs=coeffs.default
Tag=6126 max_avx512=2.3 max_avx2=2.9 ceffs=coeffs.6126.default max_power=600 error_power=700
      idle_governor=ondemand
```

11.2.10 Power policies plug-ins

```
# Policy names must be exactly file names for policies installeled in the system.
DefaultPowerPolicy=monitoring
Policy=monitoring Settings=0 DefaultFreq=2.4 Privileged=0
Policy=min_time Settings=0.7 DefaultFreq=2.0 Privileged=0
Policy=min_energy Settings=0.05 DefaultFreq=2.4 Privileged=1

# For homogeneous systems, default frequencies can be easily specified using freqs.
# For heterogeneous systems it is preferred to use pstates.

# Example with pstates (lower pstates corresponds with higher frequencies).
# Pstate=1 is nominal and 0 is turbo
#Policy=monitoring Settings=0 DefaultPstate=1 Privileged=0
#Policy=min_time Settings=0.7 DefaultPstate=4 Privileged=0
#Policy=min_energy Settings=0.05 DefaultPstate=1 Privileged=1

# Tags can be also used with policies for specific configurations
#Policy=monitoring Settings=0 DefaultFreq=2.6 Privileged=0 tag=6126
```

11.2.11 Island description

This section is mandatory since it is used for cluster description. Normally nodes are grouped in islands that share the same hardware characteristics as well as its database managers (EARDBDS). Each entry describes part of an island, and every node must be in an island.

There are two kinds of database daemons. One called **server** and other one called **mirror**. Both perform the metrics buffering process, but just one performs the insert. The mirror will do that insert in case the 'server' process crashes or the node fails.

It is recommended for all islands to maintain server-mirror symmetry. For example, if the island I0 and I1 have the server N0 and the mirror N1, the next island would have to point the same N0 and N1 or point to new ones N2 and N3, not point to N1 as server and N0 as mirror.

Multiple EARDBDs are supported in the same island, so more than one line per island is required, but the condition of symmetry have to be met.

It is recommended that for an island the server and the mirror to be running in different nodes. However, the EARDBD program could be both server and mirror at the same time. This means that the islands I0 and I1 could have the N0 server and the N2 mirror, and the islands I2 and I3 the N2 server and N0 mirror, fulfilling the symmetry requirements.

A tag can be specified that will apply to all the nodes in that line. If no tag is defined, the default one will be used as hardware definition.

Finally, if an EARGM is being used to cap power, the EARGMID field is necessary in at least one line, and will specify what EARGM controls the nodes declared in that line. If no EARGMID is found in a line, the first one found will be used (ie, the previous line EARGMID).

```
# In the following example the nodes are clustered in two different islands,
# but the Island 1 have two types of EARDBDs configurations.

Island=0 DBIP=node1081 DBSECIP=node1082 Nodes=node10[01-80] EARGMID=1

# These nodes are in island0 using different DB connections and with a different architecture

Island=0 DBIP=node1084 DBSECIP=node1085 Nodes=node11[01-80] DBSECIP=node1085 tag=6126

# These nodes are in island0 and will use default values for DB connection (line 0 for island0) and default
      tag
#These nodes will use the same EARGMID as the previous ones
Island=0 Nodes=node12[01-80]

# Will use default tag
Island=1 DBIP=node1181 DBSECIP=node1182 Nodes=node11[01-80]
```

Detailed island accepted values:

- nodename_list accepts the following formats:

  - Nodes=node1,node2,node3
  - Nodes=node\\\[1-3\\\]
  - Nodes=node\\\[1,2,3\\\]

- Any combination of the two latter options will work, but if nodes have to be specified individually (the first format) as of now they have to be specified in their own line. As an example:
  - Valid formats:
    * Island=1 Nodes=node1,node2,node3
    * Island=1 Nodes=node\\\[1-3\\\],node\\\[4,5\\\]
  - Invalid formats:
    * Island=1 Nodes=node\\\[1,2\\\],node3
    * Island=1 Nodes=node\\\[1-3\\\],node4

11.2.12 EDCMON

This section specifes the list of sensors, types, pdu ips etc for the edcmon. Even though he edcmon includes other plugins for testing purposes, the main goal is the data center monitor so this section only addresses this use case.

```
# EDCMON section
# sensor_list field must be placed at the end. It is a comma separated list of sensors names.
# Use quotes "" to group sensors lists or sensor names includig spaces
# host can be any or a hostname
# pdu_type can be : storage, management, network or others
#
edcmontag=stg pdu_type=storage pdu_ips=pduip1,pduip2,pduip3 sensor_list="Internal Humidity,Internal
      Temperature,Total Real Power"
edcmontag=ntw pdu_type=network pdu_ips=pduip4 sensor_list="Internal Humidity,Internal Temperature,Total Real
      Power"
edcmontag=mgt pdu_type=management pdu_ips=pdu5 sensor_list="Power"
edcmontag=spe host=host1 pdu_type=others pdu_ips=pdu6 sensor_list=Power
```

11.3 SLURM SPANK plug-in configuration file

SLURM loads the plug-in through a file called `plugstack.conf`, which is composed by a list of a plug-ins. In the file `etc/slurm/ear.plugstack.conf`, there is an example entry with the paths already set to the plug-in, temporal and configuration paths.

Example:

```
required ear_install_path/lib/earplug.so  prefix=ear_install_path sysconfdir=etc_ear_path
      localstatedir=tmp_ear_path earlib_default=off
```

The argument `prefix` points to the EAR installation path and it is used to load the library using `LD_PRELOAD` mechanism. Also the `localstatedir` is used to contact with the EARD, which by default points the path you set during the `./configure` using `--localstatedir` or `EAR_TMP` arguments. Next to these fields, there is the field `earlib_default=off`, which means that by default EARL is not loaded. Finally there are `eargmd_host` and `eargmd_port` if you plan to connect with the EARGMD component (you can leave this empty).

Also, there are two additional arguments. The first one, `nodes_allowed=` followed by a comma separated list of nodes, enables the plug-in only in that nodes. The second, `nodes_excluded=`, also followed by a comma separated list of nodes, disables the plug-in only in nodes in the list. These are arguments for very specific configurations that must be used with caution, if they are not used it is better that they are not written.

Example:

```
required ear_install_path/lib/earplug.so  prefix=ear_install_path sysconfdir=etc_ear_path
      localstatedir=tmp_ear_path earlib_default=off nodes_excluded=node01,node02
```

11.4 MySQL/PostgreSQL

WARNING: If any EAR component is running in the same machine as the MySQL server some connection problems might occur. This will not happen with PostgreSQL. To solve those issues, input into MySQL's CLI client the `CREATE USER` and `GRANT PRIVILEGES` queries from `edb_create -o` changing the portion ''user_↩
name'@''`to`'user_name'@'localhost'` so that EAR's users have access to the server from the local machine. There are two ways to configure a database server for EAR's usage.

- Run `edb_create -r` located in `$EAR_INSTALLATION_PATH/sbin` from a node with root access to the MySQL server. This requires MySQL/PostgreSQL's section of ear.conf to be correctly written. For more info run `edb_create -h`.
- Manually create the database and users specified in ear.conf, as well as the required tables. If ear.conf has been configured, running `edb_create -o` will output the queries that would be run with the program that contain all that is needed for EAR to properly function.

For more information about how each `ear.conf` flag changes the database creation, see our [Database section](EAR-Database). For further information about EAR's database management tools, see the Commands section.

11.5 MSR Safe

MSR Safe is a kernel module that allows to read and write MSR without root permission. EAR opens MSR Safe files if the ordinary MSR files fail. MSR Safe requires a configuration file to allow read and write registers. You can find configuration files in `etc/msr_safe` for Intel Skylake and superior and AMD Zen and superior.

You can pass these configuration files to MSR Safe kernel mode like this:
```
cat intel63 > /dev/cpu/msr_allowlist
```

You can find more information in the `official repository`.

Chapter 12

Learning-phase

This is a necessary phase prior to the normal EAR utilization and is a kind of hardware characterization of the nodes. During the phase a matrix of coefficients are calculated and stored. These coefficients will be used to predict the energy consumption and performance of each application.

Please, visit the learning phase `wiki page` to read the manual and the `repository` to get the scripts and the kernels.

12.1 Tools

The following table lists tools provided with EAR package to work with coefficients computed during the learning phase.

Name	Description	Basic arguments
coeffs_compute	Computes the learning coefficients.	<save path> <min_freq> <nodename>
coeffs_default	Computes the default coefficients file.	
coeffs_null	Creates a dummy configuration file to be used by EARD.	<coeff_path>, <max_freq> <min_freq>
coeffs_show	Shows the computed coefficients file in text format.	<file_path>

Use the argument `--help` to expand the application information and list the admitted flags.

12.1.1 Examples

Compute the coefficients for the node `node1001` in which the minimum frequency set during the learning phase was 1900000 KHz

```
./coeffs_compute /etc/coeffs 1900000 node1001
```

Chapter 13

EAR plug-ins

Some of the core of EAR functionality can be dynamically loaded through a plug-in mechanism, making EAR more extensible and dynamic than previous versions since it is not needed to reinstall the system to add, for instance, a new policy or a new power model. It is only needed to copy the file in the `$EAR_INSTALL_PATH/lib/plugins` folder and restart some components. The following table lists the current EAR functionalities designed with a plug-in mechanism:

Plug-in	Description
Power model	Energy models used by energy policies.
Power policies	Energy policies themselves.
Energy readings	Node energy readings.
Tracing	Execution traces.
Report	[Data reporting](Report).
Powercap	Powercap management.

13.1 Considerations

- Plug-in **paths** is set by default to `$EAR_INSTALL_PATH/lib/plugins`.
- Default **power model** library is specified in `ear.conf` (*energy_model* option). By default EAR includes a `basic_model.so` and `avx512_model.so` plug-ins.
- The **node energy readings** library is specified at `ear.conf` in the *energy_plugin* option for each tag. Several plug-ins are included: `energy_nm.so` (uses Intel NodeManager IPMI commands), `energy_rapl.so` (uses a node energy estimation based on DRAM and PACKAGE energy provided by RAPL), `energy_sd650.so` (uses the high frequency IPMI hardware included in Lenovo SD650 systems) and the `energy_inm_power_freeipmi.so`, which uses the Intel Node Manager power reading commands and requires the freeipmi library.
- **Power policies** included in EAR are: `monitoring.so`, `min_energy.so`, `min_time.so`, `min_energy_no_models.so` and `min_time_no_models.so`. The list of policies installed is automatically detected by the EAR plug-in. However, only policies included in `ear.conf` can be used.
- The **tracing** is an optional functionality. It is included to provide additional information or to generate runtime information.
- **Report** plug-ins include different options to report EAR data from the different components. By default it is included the eard, eardbd, csv_ts, mysql/psql (depending on the installation). Plug-ins to be loaded by default can be specified on the `ear.conf`. For more information, check the [report section](Report)

  Note SLURM Plugin does not fit in this philosophy, it is a core component of EAR and can not be replaced by any third party development.

Chapter 14

EAR Powercap

EAR provides powercap at different levels:

- Node powercap, where a node cannot exceed their given power consumption.
- Cluster powercap, where the target power is for the entire cluster. It uses the node powercap to achieve its target.

14.1 Node powercap

Node powercap is enforced by the EARD. The initial values for each node's powercap are set in the tags section of the ear.conf (see Tags for more information), which include the power limit, the CPU/PKG powercap plugin and the GPU powercap plugin (if needed). The power limit can be changed at runtime via `econtrol` or by an active `EARGM` that has the node under its control.

The EARD enforces the powercap via its plugins, which in turn ensure that the domain they control (CPU/GPU) does not exceed their power allocation.

The main goals of the node powercap is, first and foremost, to enforce the power limit with the secondary goal to maximize performance while under said limit. The EARD will use its current power limit as a budget which it will, in turn, distribute among the domains (controlled by the plugins) according to the current node's needs.

Node powercap can be applied without cluster powercap by defining only the node powercap in the EAR configuration file.

14.2 Cluster powercap

Cluster powercap is managed by one or more EARGMs and enforced at a node level by the EARD. EARGMs have an individual power limit set in their definition (see EARGM for more details) and the monitoring frequency. Each EARGM will then ask the nodes under its control (as indicated in the nodes' definition for its power consumption and distribute the budget accordingly. There are two main ways in which the cluster powercap might be enforced; soft and hard cluster powercap.

14.2.1 Soft cluster powercap

This type of powercap is targeted to systems where exceeding the power limit is not a hardware constraint but a rule that needs enforcement for a different reason. In this scenario, the compute nodes will run as if no limit was applied until the total power consumption of the cluster reaches a percentage threshold (defined as the suspend threshold in ear.conf), at which point the EARGM will send a power limit to all the nodes to prevent the global power to go above the actual limit. Additionally, a script can be attached to the activation of the powercap in which the admin can set whichever actions they feel appropriate. Once the cluster power goes below another percentage threshold (defined as the resume threshold in ear.conf) the EARGM will send a message to all the nodes to go back to unlimited power usage, as well as call the deactivation script set by the admin (if any is specified).

In terms of configuration, `EARGMPowerCapMode` must be set to 2 (soft powercap) and all nodes need to have a `max_powercap` set in their tag. The value of `max_powercap` will be the power allocation of the nodes that have that tag. If a node has a `max_powercap` value of 1, 0 or -1 they will ignore powercap messages from an EARGM in soft cluster powercap mode.

14.2.2 Hard cluster powercap

Hard powercap is used when the system must not, under any circumstance, go above the power limit. This starts by always having a set powercap in the compute nodes. The job of the EARGM is to periodically monitor the state of the nodes, which will request more or less power depending on their current workload, and redistribute the power according to the needs of all nodes.

14.3 Possible powercap values

To set the powercap for an entire cluster one can do it two ways, specific values and calculated. With specific values, the `powercap` value in the EARGM definition must be a number > 0, and that will be the power budget for the EARGM to distribute among the nodes it controls. On the other hand, if `powercap=-1` the total power budget will be calculated automatically as the sum of the powercap values set in the tags for the nodes it controls.

For an EARD, the valid values of `powercap` in its tag are 1 and N > 1. When set to 1, the daemon will run with no power limit until it receives one. On the other hand, if the powercap is a higher number that will be used as the power limit until a different value is set via `econtrol` or EARGM reallocations.

If either powercap or EARGMPowercapMode is set to 0 in the configuration file, the thread that controls the power limits will not be started and the feature will be disabled.

If the initial powercap value for a node is set to 0 the powercap will be disabled for that node and it will ignore any attempts to set it to a certain value. Set it to 1 if you ever want to set the powercap.

14.4 Example configurations

The following is an example for hard powercap on 4 nodes, with a starting powercap of 225W each and a total power budget of 1000W. For clarity a few fields in the tags section have been skipped.

```
# Wait period between power checks
EARGMPowerPeriod=120
# Activate powercap
EARGMPowercapMode=1
# Set up at least 1 EARGM
EARGMId=1 energy=XXX power=1000 node=node1

# Set up the nodes
```

```
Tag=tag1 default=yes max_power=500 min_power=50 error_power=600 powercap=225 powercap_plugin=dvfs.so
      gpu_powercap_plugin=gpu.so

Island=1 nodes=node[1-4] EARGMId=1
```

This example is similar to the previous one, but the global powercap is calculated by the EARGM as the sum of the nodes. In this case, the nodes start with a default powercap of 250W and the total budget for the cluster remains 1000W.

```
# Wait period between power checks
EARGMPowerPeriod=120
# Activate powercap
EARGMPowercapMode=1
# Set up at least 1 EARGM
EARGMId=1 energy=XXX power=-1 node=node1

# Set up the nodes
Tag=tag1 default=yes max_power=500 min_power=50 error_power=600 powercap=250 powercap_plugin=dvfs.so
      gpu_powercap_plugin=gpu.so

Island=1 nodes=node[1-4] EARGMId=1
```

The following is a soft powercap example with a power budget of 1000W. The nodes will start without a set powercap but will be ready to activate it.

```
# Wait period between power checks
EARGMPowerPeriod=120
# Activate powercap as soft powercap
EARGMPowercapMode=2
# Set up at least 1 EARGM
EARGMId=1 energy=XXX power=1000 node=node1

# Set up the nodes
Tag=tag1 default=yes max_power=500 min_power=50 error_power=600 powercap=1 powercap_plugin=dvfs.so
      gpu_powercap_plugin=gpu.so

Island=1 nodes=node[1-4] EARGMId=1
```

Finally, this example has ONLY node powercap, with the nodes having a limit of 250W. There will be no reallocation:

```
# Wait period between power checks
EARGMPowerPeriod=120
# Activate powercap
EARGMPowercapMode=1
# Set up at least 1 EARGM
EARGMId=1 energy=XXX power=0 node=node1

# Set up the nodes
Tag=tag1 default=yes max_power=500 min_power=50 error_power=600 powercap=250 powercap_plugin=dvfs.so
      gpu_powercap_plugin=gpu.so

Island=1 nodes=node[1-4] EARGMId=1
```

This is the same, but deactivating the powercap by setting the mode to 0:

```
# Wait period between power checks
EARGMPowerPeriod=120
# Activate powercap
EARGMPowercapMode=0
# Set up at least 1 EARGM
EARGMId=1 energy=XXX power=1000 node=node1

# Set up the nodes
Tag=tag1 default=yes max_power=500 min_power=50 error_power=600 powercap=250 powercap_plugin=dvfs.so
      gpu_powercap_plugin=gpu.so

Island=1 nodes=node[1-4] EARGMId=1
```

This is an erroneous way to set it up, because the nodes' powercap capabilities will not be active:

```
# Wait period between power checks
EARGMPowerPeriod=120
# Activate powercap
EARGMPowercapMode=1
# Set up at least 1 EARGM
EARGMId=1 energy=XXX power=1000 node=node1

# Set up the nodes
Tag=tag1 default=yes max_power=500 min_power=50 error_power=600 powercap=0 powercap_plugin=dvfs.so
      gpu_powercap_plugin=gpu.so

Island=1 nodes=node[1-4] EARGMId=1
```

Similarly, this following example does not work because the EARGM cannot calculate a valid powercap when the nodes are set to unlimited:

```
# Wait period between power checks
EARGMPowerPeriod=120
# Activate powercap
EARGMPowercapMode=1
# Set up at least 1 EARGM
EARGMId=1 energy=XXX power=-1 node=node1

# Set up the nodes
Tag=tag1 default=yes max_power=500 min_power=50 error_power=600 powercap=1 powercap_plugin=dvfs.so
      gpu_powercap_plugin=gpu.so

Island=1 nodes=node[1-4] EARGMId=1
```

14.5 Valid configurations

There are three special values for powercap configuration, 1 (unlimited, only for Tags/Node), 0 (disabled) and -1 (auto-configure).

Furthermore, there are three cluster powercap modes for EARGM: 0 (monitoring-only), 1 (hard cluster powercap) and 2 (soft cluster powercap).

EARGM powercap mode	EARGM powercap value	Tag powercap value	Result
ANY	0	1	Cluster powercap disabled, node powercap unlimited (but can be set with `econtrol`)
ANY	0	0	All powercap types disabled, and cannot be modified without restarting
ANY	0	N	Cluster powercap disabled, node powercap set to N
HARD	-1	N	Cluster powercap set to the sum of the nodes' powercap. Node powercap set to N
HARD	N	-1	Cluster powercap set to N. Node powercap set to N/number of nodes controlled by EARGM
HARD	N	M	Cluster powercap set to N. Node powercap set to N
SOFT	N	1	Cluster powercap set to N, node powercap unlimited. If triggered, node powercap will be set to their max_powercap value
SOFT	N	M	∗ERROR ∗
HARD/SOFT	N	0	*ERROR*
HARD/SOFT	-1	-1	*ERROR*
HARD/SOFT	0	-1	∗ERROR
HARD/SOFT	1	-1	∗ERROR
HARD/SOFT	-1	1	*ERROR*

`NOTE: When using soft cluster powercap, max_powercap value must be properly set for the powercap to work.`

Chapter 15

Report

EAR reporting system is designed to fit any requirement to store all data collected by its components. By this way, EAR includes several report plug-ins that are used to send data to various services.

15.1 Overview

The reporting system is implemented by an internal API used by EAR components to report data at specific events/stages, and the report plug-in used by each one can be set at `ear.conf` file. The `Node Manager`, the `Database Manager`, the `Job Manager` and the `Global Manager` are those configurable components. The EAR Job Manager differs from other components since it lets the user to choose other plug-ins at job submission time. Check out how at the `Environment variables` section.

Plug-ins are compiled as shared objects and are located at `$EAR_INSTALL_PATH/lib/plugins/report`. Below there is a list of the report plug-ins distributed with the official EAR software.

Report plug-in name	Description
eard.so	Reports data to the EAR Node Manager. Then, it is up to the daemon to report the data as it was configured. This plug-in was mainly designed to be used by the EAR Job Manager.
eardbd.so	Reports data to the EAR Database Manager. Then, it is up to this service to report the data as it was configured. This plug-in was mainly designed to be used by the EAR Node Manager.
mysql.so	Reports data to a MySQL database using the official C bindings. This plug-in was first designed to be used by the EAR Database Manager.
psql.so	Reports data to a PosgreSQL database using the official C bindings. This plug-in was first designed to be used by the EAR Database Manager.
prometheus.so	This plug-in exposes system monitoring data in OpenMetrics format, which is fully compatible with Prometheus.
examon.so	Sends application accounting and system metrics to EXAMON.
dcdb.so	Sends application accounting and system metrics to DCDB.
sysfs.so	Exposes system monitoring data through the file system.
csv_ts.so	Reports loop and application data to a CSV file. It is the report plug-in loaded when a user sets `--ear-user-db` flag at submission time.

15.2 Prometheus report plugin

15.2.1 Requirements

The Prometheus plugin has only one dependency, microhttpd. To be able to compile it make sure that it is in your LD_LIBRARY_PATH.

15.2.2 Installation

Currently, to compile and install the prometheus plugin one has the run the following command.

```
make FEAT_DB_PROMETHEUS=1
make FEAT_DB_PROMETHEUS=1 install
```

With that, the plugin will be correctly placed in the usual folder.

15.2.3 Configuration

Due to the way in which Prometheus works, this plugin is designed to be used by the EAR Daemons, although the EARDBD should not have many issues running it too.

To have it running in the daemons, simply add it to the corresponding line in the [configuration file](Configuration).

```
EARDReportPlugins=eardbd.so:prometheus.so
```

This will expose the metrics on each node on a small HTTP server. You can access them normally through a browser at port 9011 (fixed for now).

In Prometheus, simply add the nodes you want to scrape in prometheus.yml with the port 9011. Make sure that the scrape interval is equal or shorter than the insertion time (NodeDaemonPowermonFreq in ear.conf) since metrics only stay in the page for that duration.

15.3 Examon

ExaMon (Exascale Monitoring) is a lightweight monitoring framework for supporting accurate monitoring of power/energy/thermal and architectural parameters in distributed and large-scale high-performance computing installations.

15.3.1 Compilation and installation

To compile the EXAMON plugin you need a functioning EXAMON installation.

Modify the main Makefile and set FEAT_EXAMON=1. In src/report/Makefile, update EXAMON_BASE with the path to the current EXAMON installation. Finally, set an examon.conf file somewhere on your installation, and modify src/report/examon.c (line 83, variable `char* conffile = "/hpc/opt/ear/etc/ear/examon.↩ conf"`) to point to the new examon.conf file.

The file should look like this:

```
[MQTT]

brokerHost = hostip

brokerPort = 1883

topic = org/bsc

qos = 0

data_topic_string = plugin/ear/chnl/data

cmd_topic_string = plugin/ear/chnl/cmd
```

Where `hostip` is the actual ip of the node.

Once that is set up, you can compile EAR normally and the plugin will be installed in the `lib/plugins/report` folder inside EAR's installation. To activate it, set it as one of the values in the `EARDReportPlugins` of `ear.conf` and restart the EARD.

The plugin is designed to be used locally in each node (EARD level) together with EXAMON's data broker.

15.4 DCDB

The Data Center Data Base (DCDB) is a modular, continuous, and holistic monitoring framework targeted at HPC environments.

This plugin implements the functions to report periodic metrics, report loops, and report events.

When the DCDB plugin is loaded the collected EAR data per report type are stored into a shared memory which is accessed by DCDB ear sensor (report plugin implemented on the DCDB side) to collect the data and push them into the database using MQTT messages.

15.4.1 Compilation and configuration

This plugin is automatically installed with the default EAR installation. To activate it, set it as one of the values in the `EARDReportPlugins` of `ear.conf` and restart the EARD.

The plugin is designed to be used locally in each node (EARD level) with the DCDB collect agent.

15.5 Sysfs Report Plugin

This is a new report plugin to write EAR collected data into a file. Single file is generated per metric per jobID & stepID per node per island per cluster. Only the last collected data metrices are stored into the files, means every time the report runs it saves the current collected values by overwriting the pervious data.

15.5.1 Namespace Format

The below schema has been followed to create the metric files:

```
/root_directory/cluster/island/nodename/avg/metricFile
/root_directory/cluster/island/nodename/current/metricFile
/root_directory/cluster/island/jobs/jobID/stepID/nodename/avg/metricFile
/root_directory/cluster/island/jobs/jobID/stepID/nodename/current/metricFile
```

The root_directory is the default path where all the created metric files are generated.

The cluster, island and nodename will be replaced by the island number, cluster name, and node information.

`metricFile` will be replaced by the name of the metrics collected by EAR.

15.5.2 Metric File Naming Format

The naming format used to create the metric files is implementing the standard sysfs interface format. The current commonly used schema of file naming is <type>_<component>_<metric-name>_<unit>.

Numbering is used with some metric files if the component has more than one instance like FLOPS counters or GPU data. Examples of some generated metric files:

- dc_power_watt
- app_sig_pck_power_watt
- app_sig_mem_gbs
- app_sig_flops_6
- avg_imc_freq_KHz

15.5.3 Metrics reported

The following are the reported values for each type of metric recorded by ear:

- report_periodic_metrics
  - Average values
    - The frequency and temperature values have been calculated by summing the values of all periods since the report loaded until the current period and divide it by the total number of periods.
    - The energy value is accumulated value of all the periods since the report loaded until the current one.
    - The path to those metric files built as: /root_directory/cluster/island/nodename/avg/metricFile
- Current values
  - Represent the current collected EAR metric per period.
  - The path to those metric files built as: /root_directory/cluster/island/nodename/current/metricFile
- report_loops
  - Current values
    - Represent the current collected EAR metric per loop.
    - The path to those metric files built as: /root_directory/cluster/island/jobs/jobID/step↩ ID/nodename/current/metricFile
- report_applications
  - Current values
    - Represent the current collected EAR metric per application.
    - The path to those metric files built as: /root_directory/cluster/island/jobs/jobID/step↩ ID/nodename/avg/metricFile
- report_events
  - Current values
    - Represent the current collected EAR metric pere event.
    - The path to those metric files built as: /root_directory/cluster/island/jobs/jobID/step↩ ID/nodename/current/metricFile

Note: If the cluster contains GPUs, both report_loops and report_applications will generate new schema files will per GPU which contain all the collected data for each GPU with the paths below:

- /root_directory/cluster/island/jobs/jobID/stepID/nodename/current/GPU-ID/metricFile
- /root_directory/cluster/island/jobs/jobID/stepID/nodename/avg/GPU-ID/metricFile

15.6 CSV

This plug-in reports both application and loop signatures in CSV format. Note that the latter can only be reported if the application is running with the EAR Job Manager. Fields are separated by semi-colons (i.e., ;). This plug-in is the one loaded by default when a user sets `--ear-user-db` submission flag.

By default output files are named `ear_app_log.<nodename>.time.csv` and `ear_app_log.<nodename>.time.loops.csv` for applications and loops, respectively. This behaviour can be changed by exporting `EAR_USER_DB_PATHNAME` environment variable. Therefore, output files are `<env var value>.<nodename>.time.csv` for application signatures and `<env var value>.<nodename>.time.loops.csv` for loop signatures.

> When setting `--ear-user-db=something` flag at submission time, the batch scheduler plug-in sets this environment variable for you.

The following table describes **application signature file fields**:

Field	Description	Format
NODENAME	The short node name the following signature belongs to.	string
JOBID	The Job ID the following signature belongs to.	integer
STEPID	The Step ID the following signature belongs to.	integer
APPID	The Application ID the following signature belongs to.	integer
USERID	The user owning the application.	string
GROUPID	The main group the user owning the application belongs to.	string
JOBNAME	The name of the application being runned. In SLURM systems, this value honours `SLURM_JOB_NAME` environment variable. Otherwise, it is the executable program name.	string
USER_ACC	This is the account of the user which ran the application. Only supported in SLURM systems.	string
ENERGY_TAG	The energy tag requested with the application (see `ear.conf`).	string
POLICY	The Job Manager optimization policy executed (if applies).	string
POLICY_TH	The power policy threshold used (if applies).	real
START_TIME	The timestamp of the beginning of the application, expressed in seconds since EPOCH.	integer
END_TIME	The timestamp of the application ending, expressed in seconds since EPOCH.	integer
START_DATE	The date of the beginning of the application, expressed in %+4Y-m-d X.	string
END_DATE	The date of the application ending, expressed in %+4Y-m-d X.	string
AVG_CPUFREQ_KHZ	The average CPU frequency across all CPUs used by the application, in kHz.	integer
AVG_IMCFREQ_KHZ	The average IMC frequency during the application execution, in kHz.	integer
DEF_FREQ_KHZ	The default CPU frequency set at the start of the application, in kHz.	integer
TIME_SEC	The total execution time of the application, in seconds.	integer
CPI	The Cycles per Instruction retrieved across all application processes.	real
TPI	Transactions to the main memory per Instruction retrieved .	real
MEM_GBS	The memory bandwidth of the application, in GB/s.	real
IO_MBS	The accumulated I/O bandwidth of the application processes, in MB/s.	real
PERC_MPI	The average percentage of time spent in MPI calls across all application processes, in %.	real

Field	Description	Format
DC_NODE_POWER_W	The average DC node power consumption in the node consumed by the application, in Watts.	real
DRAM_POWER_W	The average DRAM power consumption in the node consumed by the application.	real
PCK_POWER_W	The average package power consumption in the node consumed by the application	real
CYCLES	The total cycles consumed by the application, accumulated across all its processes.	integer
INSTRUCTIONS	The total number of instructions retrieved, accumulated across all its processes.	integer
CPU-GFLOPS	The total number of GFLOPS retrieved, accumulated across all its processes.	real
GPU∗i∗_POWER_W	The average power consumption of the ∗i∗th GPU in the node.	real
GPU∗i∗_FREQ_KHZ	The average frequency of the ∗i∗th GPU in the node.	real
GPU∗i∗_MEM_FREQ_KHZ	The average memory frequency of the ∗i∗th GPU in the node.	real
GPU∗i∗_UTIL_PERC	The average GPU *i* utilization.	integer
GPU∗i∗_MEM_UTIL_PERC	The average GPU *i* memory utilization.	integer
GPU∗i∗_GFLOPS	The total GPU *i* GFLOPS retrieved during the application execution.	real
GPU∗i∗_TEMP	The average temperature of the ∗i∗th GPU of the node, in celsius.	real
GPU∗i∗_MEMTEMP	The average memory temperature of the ∗i∗th GPU of the node, in celsius.	real
L1_MISSES	The total numer of L1 cache misses during the application execution.	integer
L2_MISSES	The total numer of L2 cache misses during the application execution.	integer
L3_MISSES	The total numer of L3 cache misses during the application execution.	integer
SPOPS_SINGLE	The total number of floating point operations, accumulated across all processes, retrieved during the application execution.	integer
SPOPS_128	The total number of AVX128 floating point operations, accumulated across all processes, retrieved during the application execution.	integer
SPOPS_256	The total number of AVX256 floating point operations, accumulated across all processes, retrieved during the application execution.	integer
SPOPS_512	The total number of AVX512 floating point operations, accumulated across all processes, retrieved during the application execution.	integer
DPOPS_SINGLE	The total number of double precision floating point operations, accumulated across all processes, retrieved during the application execution.	integer
DPOPS_128	The total number of double precision AVX128 floating point operations, accumulated across all processes, retrieved during the application execution.	integer
DPOPS_256	The total number of double precision AVX256 floating point operations, accumulated across all processes, retrieved during the application execution.	integer
DPOPS_512	The total number of double precision AVX512 floating point operations, accumulated across all processes, retrieved during the application execution.	integer
TEMP∗i∗	The average temperature of the socket *i* during the application execution, in celsius.	real

Chapter 16

EAR Database

16.1 Tables

16.1.1 Application information

The following tables contain information directly related to applications executed on the system while EAR was monitoring. The main key is the JOBID.STEPID combination generated by the scheduler.

- **Jobs**: job information (app_id, user_id, job_id, step_id, etc). One record per JOBID.STEPID is created in the DB.
- **Applications**: this table's records serve as a link between Jobs and Signatures, providing an application signature (from EARL) for each node of a job. One record per JOBID.STEPID.NODENAME is created in the DB.
- **Loops**: similar to *Applications*, but stores a Signature for each application loop detected by EARL, instead of one per each application. This table provides internal details of running applications and could significantly increase the DB size.
- **Signatures**: EARL computed signature and metrics. One record per JOBID.STEPID.NODENAME is created in the DB when the application is executed with EARL.
- **GPU_signatures**: EARL computed GPU signatures. This information belongs to a loop or application signature. If the signature is from a node with 4 GPUs there will be 4 records.
- **Power_signatures**: Basic time and power metrics that can be obtained without EARL. Reported for all applications. One record per JOBID.STEPID.NODENAME is created in the DB.

16.1.2 System monitoring

This tables contain periodic information gathered from the nodes. There is a single-node information table and an aggregated one to increase the speed of queries to get cluster-wide information.

- **Periodic_metrics**: node metrics reported every N seconds (N is defined in `ear.conf`).
- **Periodic_aggregations**: sum of all *Periodic_metrics* in a time period to ease accounting in `ereport` command and EARGM, as well as reducing database size (*Periodic_metrics* of older periods where precision at node level is not needed can be deleted and the aggregations can be used instead).

16.1.3 Events

- **Events**: EAR events report. There are several types of events, depending on their source: EARL, EARD-powercap, EARD-runtime and EARGM. For more information, see the table's fields and its header file (src/common/types/event_type.h). For EARL-specific events, also see this.

16.1.4 EARGM reports

- **Global_energy**: contains reports of cluster-wide energy accounting set by EARGM using the parameters in `ear.conf`. One record every T1 period (defined at ear.conf) is reported.

16.1.5 Learning phase

This tables are the same as their non-learning counterparts, but are specifically used to store the applications executed during a learning phase.

- **Learning_applications**: same as *Applications*, restricted to learning phase applications.
- **Learning_jobs**: same as *Jobs*, restricted to learning phase jobs.
- **Learning_signatures**: same as *Signatures*, restricted to learning phase job metrics.

  NOTE In order to have *GPU_signatures* table created and *Periodic_metrics* containing GPU data, the databasease must be created (if you follow the `edb_create` approach, see the section down below) with GPUs enabled at the compilation time. See how to update from previous versions if you are updating EAR from a release not having GPU metrics.

16.2 Creation and maintenance

To create the database a command (`edb_create`) is provided by EAR, which can either create the database directly or provide the queries for the database creation so the administrator can use them or modify them at their discretion (any changes may alter the correct function of EAR's accounting).

Since a lot of data is reported by EAR to the database, EAR provides two commands to remove old data and free up space. These are intended to be used with a `cron` job or a similar tool, but they can also be run manually without any issues. The two tools are `edb_clean_pm` to remove periodic data accounting from nodes, and `edb_clean_apps` to remove all the data related to old jobs.

For more information on this commands, check the commands' page on the wiki.

16.3 Database creation and ear.conf

When running `edb_create` some tables might not be created, or may have some quirks, depending on some `ear.conf` settings. The settings and alterations are as follows:

- `DBReportNodeDetail`: if set to 1, `edb_create` will create two additional columns in the *Periodic_metrics* table for Temperature (in Celsius) and Frequency (in Hz) accounting.
- `DBReportSigDetail`: if set to 1, *Signatures* will have additional fields for cycles, instructions, and FLOPS1-8 counters (number of instruction by type).
- `DBMaxConnections`: this will restrict the number of maximum simultaneous commands connections.

If any of the settings is set to 0, the table will have fewer details but the table's records will be smaller in stored size.

Any table with missing columns can be later altered by the admin to include said columns. For a full detail of each table's columns, run `edb_create -o` with the desired `ear.conf` settings.

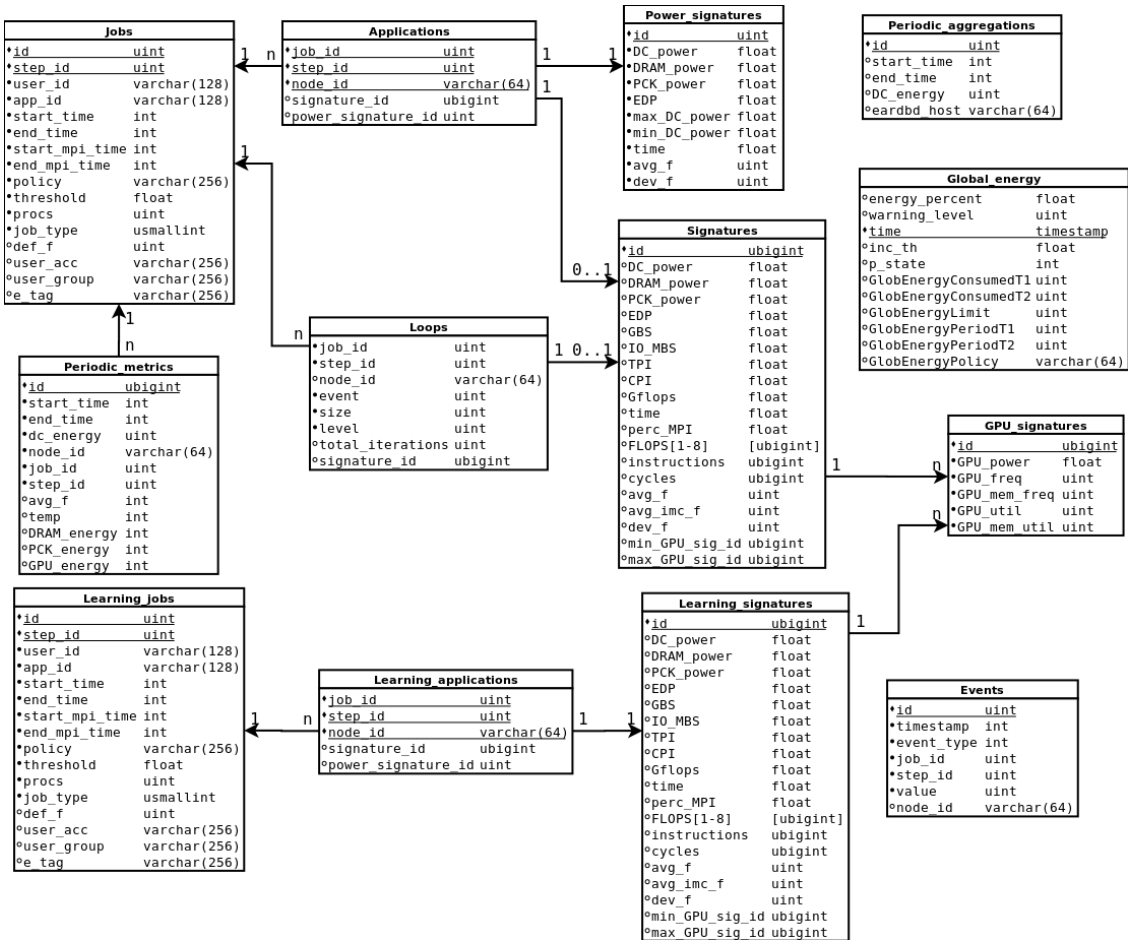
16.4 Information reported and ear.conf

There are various settings in `ear.conf` that restrict data reported to the database and some errors might occur if the database configuration is different from EARDB's.

- `DBReportNodeDetail`: if set to 1, node managers will report temperature, average frequency, DRAM and PCK energy to the database manager, which will try to insert it to *Periodic_metrics*. If *Periodic_metrics* does not have the columns for both metrics, an error will occur and nothing will be inserted. To solve the error, set `ReportNodeDetail` to 0 or manually update *Periodic_metrics* in order to have the necessary columns.
- `DBReportSigDetail`: similarly to `ReportNodeDetail`, an error will occur if the configuration differs from the one used when creating the database.
- `DBReportLoops` : if set to 1, EARL detected application loops will be reported to the database, each with its corresponding Signature. Set to 0 to disable this feature. Regardless of the setting, no error should occur.

If *Signatures* and/or *Periodic_metrics* have additional columns but their respective settings are set to 0, a NULL will be set in those additional columns, which will make those rows smaller in size (but bigger than if the columns did not exist).

Additionally, if EAR was compiled in a system with GPUs (or with the GPU flag manually enabled), another table to store GPU data will be created.



NOTE the nomenclature is modified from MySQL's type. Any type starting with `u` is unsigned. `bigint` corresponds to an integer of 64 bits, `int` is 32 and `smallint` is 16.

For a detailed description of each field in any of the database's tables, see [here](EAR-database-table-descriptions).

16.5 Updating from previous versions

16.5.1 From EAR 4.3 to 5.0

For better internal consistency, Jobs' id field was renamed:

```
ALTER TABLE Jobs CHANGE id job_id INT UNSIGNED;
```

To add support for workflows, a new field was added to several tables to allow their accounting:

```
ALTER TABLE Jobs ADD COLUMN local_id INT UNSIGNED NOT NULL AFTER step_id;
ALTER TABLE Jobs DROP PRIMARY KEY, ADD PRIMARY KEY (id, step_id, local_id);
ALTER TABLE Applications ADD COLUMN local_id INT UNSIGNED NOT NULL AFTER step_id;
ALTER TABLE Applications DROP PRIMARY KEY, ADD PRIMARY KEY (job_id, step_id, local_id);
ALTER TABLE Loops ADD COLUMN local_id INT UNSIGNED NOT NULL AFTER step_id;
```

16.5.2 From EAR 4.2 to 4.3

Three new fields corresponding to L1, L2 and L3 cache misses have been added to the signatures.

> **NOTE** This change only applies to the databases that have been created with the extended application signature (i.e. they have the FLOPS, instructions and cycles counters in their signatures).

```
ALTER TABLE Signatures
ADD COLUMN L1_misses BIGINT UNSIGNED AFTER perc_MPI,
ADD COLUMN L2_misses BIGINT UNSIGNED AFTER L1_misses,
ADD COLUMN L3_misses BIGINT UNSIGNED AFTER L2_misses;
ALTER TABLE Learning_signatures
ADD COLUMN L1_misses BIGINT UNSIGNED AFTER perc_MPI,
ADD COLUMN L2_misses BIGINT UNSIGNED AFTER L1_misses,
ADD COLUMN L3_misses BIGINT UNSIGNED AFTER L2_misses;
```

16.5.3 From EAR 4.1 to 4.2

A field in the Events table had its name changed to be more generic and its type changed to unsigned:

```
ALTER TABLE Events CHANGE freq value INT unsigned;
```

Furthermore, some errors on big servers have been found due to the ids of a few fields being too small. To correct this, please run the following commands:

```
ALTER TABLE Learning_signatures MODIFY COLUMN id BIGINT unsigned AUTO_INCREMENT;
ALTER TABLE Signatures MODIFY COLUMN id BIGINT unsigned AUTO_INCREMENT;
ALTER TABLE Applications MODIFY COLUMN signature_id BIGINT unsigned;
ALTER TABLE Loops MODIFY COLUMN signature_id BIGINT unsigned;
```

If GPUs are being used, also run:

```
ALTER TABLE GPU_signatures MODIFY COLUMN id BIGINT unsigned AUTO_INCREMENT;
ALTER TABLE Learning_signatures MODIFY COLUMN min_gpu_sig_id BIGINT unsigned;
ALTER TABLE Learning_signatures MODIFY COLUMN max_gpu_sig_id BIGINT unsigned;
ALTER TABLE Signatures MODIFY COLUMN min_gpu_sig_id BIGINT unsigned;
ALTER TABLE Signatures MODIFY COLUMN max_gpu_sig_id BIGINT unsigned;
```

16.5.4 From EAR 3.4 to 4.0

Several fields have to be added in this update. To do so, run the following commands to the database's CLI client:

```
ALTER TABLE Signatures ADD COLUMN avg_imc_f INT unsigned AFTER avg_f;
ALTER TABLE Signatures ADD COLUMN perc_MPI FLOAT AFTER time;
ALTER TABLE Signatures ADD COLUMN IO_MBS FLOAT AFTER GBS;

ALTER TABLE Learning_signatures ADD COLUMN avg_imc_f INT unsigned AFTER avg_f;
ALTER TABLE Learning_signatures ADD COLUMN perc_MPI FLOAT AFTER time;
ALTER TABLE Learning_signatures ADD COLUMN IO_MBS FLOAT AFTER GBS;
```

16.5.5 From EAR 3.3 to 3.4

If no GPUs were used and they will not be used there are no changes necessary.

If GPUs were being used, type the following commands to the database's CLI client:

```
ALTER TABLE Signatures ADD COLUMN min_GPU_sig_id BIGINT unsigned, ADD COLUMN max_GPU_sig_id BIGINT unsigned;

ALTER TABLE Learning_signatures ADD COLUMN min_GPU_sig_id BIGINT unsigned, ADD COLUMN max_GPU_sig_id BIGINT
      unsigned;
CREATE TABLE IF NOT EXISTS GPU_signatures ( id BIGINT unsigned NOT NULL AUTO_INCREMENT, GPU_power FLOAT NOT
      NULL, GPU_freq INT unsigned NOT NULL, GPU_mem_freq INT unsigned NOT NULL, GPU_util INT unsigned NOT
      NULL, GPU_mem_util INT unsigned NOT NULL, PRIMARY KEY (id));
```

If no GPUs were being used but now are present, use the previous query plus the following one:

```
ALTER TABLE Periodic_metrics ADD COLUMN GPU_energy INT;
```

16.6 Database tables description

EAR's database contains several tables, as described here. Each table contains different information, as described here:

16.6.1 Jobs

- id: Job id given by the scheduler (for example SLURM_JOBID).
- step_id: step id given by the scheduler.
- user_id: the linux username that executed the job.
- app_id: the application/job name as given by the scheduler (not necessarily the executable's name)
- start_time: timestamp of the job's[.step] start
- end_time: timestamp of the job's[.step] end
- start_mpi_time: timestamp of the beginning of application region managed by the EARL. Named MPI for historical reasons. For MPI applications timestamp of the MPI_Init execution.
- end_mpi_time: timestamp of the end of application region managed by the EARL. Named MPI for historical reasons. For MPI applications timestamp of the MPI_Finalize execution.
- policy: EAR policy name in action for the job. Can be "No Policy" if the job runs without EAR.
- threshold: threshold used by the policy to configure it's behavior. For example, the maximum performance penalty in min_energy.
- job_type:
- def_f: default CPU frequency requested by the user/job manager.
- user_acc: the account the user_id belongs to.
- user_group: the linux group name the user_id belongs to.
- e_tag: energy tag. The user can specify an energy tag to apply pre-defined CPU frequency settings.

16.6.2 Applications

- job_id: job id given by the scheduler. Used as a foreign key for Jobs.
- step_id: step id given by the scheduler. Used as a foreign key for Jobs.
- node_id: the nodename in which the application ran. The names of the nodes are trimmed at any ".", i.e., node1.at.cluster becomes node1.
- signature_id: the id (index) of the computed signature for the job on this node. If the job runs without EAR library the field will be NULL.
- power_signature_id: the id (index) of the power signature for the job on this node.

16.6.3 Signatures

All the metrics in this table refer to the period of time where the Signature is computed. Typically is 10 sec. Signatures are only reported when the application uses the EAR library.

- id: unique id generated by the database engine to be used in JOIN queries.
- DC_power: average DC node power (in Watts).
- DRAM_power: average DRAM power, including the 2 sockets (in Watts).
- PCK_power: Average CPU power, including the 2 sockets (in Watts).
- EDP: Energy Delay Product computed as (time x time x DC_power).
- GBS: Main memory bandwidth (GB/sec).
- IO_MBS: I/O read and write rate (MB/s).
- TPI: Main memory transactions per instruction.
- CPI: Cycles per instructions.
- Gflops: Giga Floating point operations, per second, generated by the application processes in the node. GFlops/sec.
- time: total execution time (in seconds)
- perc_MPI: average percentage of MPI time vs computational time in the node. Includes all the application processes in the node.
- L1_misses: L1 cache misses counter.
- L2_misses: L2 cache misses counter.
- L3_misses: L3 cache misses counter.
- FLOPS1: Floating point operations Single precision 64 bits consumed by application processes in the node.
- FLOPS2: Floating point operations Single precision 128 bits consumed by application processes in the node.
- FLOPS3 Floating point operations Single precision 256 bits consumed by application processes in the node.
- FLOPS4: Floating point operations Single precision 512 bits consumed by application processes in the node.
- FLOPS5: Floating point operations Double precision 64 bits consumed by application processes in the node.
- FLOPS6: Floating point operations Double precision 128 bits consumed by application processes in the node.
- FLOPS7: Floating point operations Double precision 256 bits consumed by application processes in the node.

- FLOPS8: Floating point operations Double precision 512 bits consumed by application processes in the node.
- instructions: total instructions executed by the application processes in the node.
- cycles: total cycles consumed by the application processes in the node.
- avg_f: average CPU frequency (includes all the cores used by the application on the node) in KHz.
- avg_imc_f: average memory frequency (includes the two sockets) in KHz.
- def_f: default CPU frequency used at the beginning of the application in KHz.
- min_GPU_sig_id: start of the range containing the GPU_signature's ids, used for JOIN queries. If an application doesn't have GPUs it will be NULL.
- max_GPU_sig_id: end of the range containing the GPU_signature's ids, used for JOIN queries. If an application doesn't have GPUs it will be NULL.

1. Each signature corresponds to either a Loop or an Application. When it's an application it is the average values for its entire runtime. For a loop, the values are the average of only the period comprised by the loop's start and end.
    1. Signatures are only reported when an application is running with EARL.
    2. The GPU signature values are inclusive, i.e. if a signature has a min_id = 1 and max_id = 3, the GPU_signatures with ids 1,2,3 will be from this application.

16.6.4 Power_signatures

Power signatures are measured and reported by the EARD and reported for all the jobs/steps/nodes. It's independent of the EAR library utilization.

- id: unique id generated by the database engine to be used in JOIN queries.
- DC_power: average DC node power (in Watts)
- DRAM_power: average DRAM power, including the 2 sockets (in Watts)
- PCK_power: Average CPU power, including the 2 sockets (in Watts)
- EDP: Energy Delay Product computed as (time x time x DC_power)
- max_DC_power: maximum DC node power registered by the EAR daemon during the application's execution (in Watts)
- min_DC_power: minimum DC node power registered by the EAR daemon during the application's execution (in Watts)
- time: total execution time (in seconds)
- avg_f: average CPU frequency (includes all the cores of the node) in KHz
- def_f: default CPU frequency used at the beginning of the application in KHz

16.6.5 GPU_signatures

- id: unique id generated by the database engine to be used in JOIN queries.
- GPU_power: average GPU power for a single GPU (in Watts)
- GPU_freq: average GPU frequency for a single GPU (in KHz)
- GPU_mem_freq: average GPU memory frequency for a single GPU (in KHz)
- GPU_util: average GPU utilisation for the reported period for a single GPU. (percentage)
- GPU_mem_util: average GPU memory utilisation for the reported period for a single GPU.(percentage)

If an application has more than 1 GPU there will be a signature for each of them.

16.6.6 Loops

Loops are only reported when the EAR library is used.

- event: loop type identificatory. It's for internal use of the EAR library. Together with size and level is used internally.
- size: loop's size as computed by DynAIS.
- level: loop's level of depth (indicative of loops inside of loops)
- job_id: job id given by the job manager. Used as a foreign key for Jobs.
- step_id: step id given by the job manager. Used as a foreign key for Jobs.
- node_id: the nodema,e in which the application ran. The names of the nodes are trimmed at any ".", i.e., node1.at.cluster becomes node1.
- total_iterations: timestamp at which the loop signature has been reported. It is named total_iterations for historical reasons.
- signature_id: the id of the computed signature for the job on this node.

1. the combination even-size-level forms the Primary Key for the table loops.
    1. Loops will always have a signature because they are only reported when EAR is used
    2. When a loop is inserted, the corresponding Job is probably not in the database yet, because Jobs are inserted only when an application finishes. JOIN queries with Jobs can only be done once an application has finished (only the current step id needs to finish, not the entire job).

16.6.7 Events

- id: unique id generated by the database engine to use as primary key.
- timestamp: registered timestamp of when the event happened (NOT when it was inserted)
- event_type: a numerical id for the type of EAR event
- job_id: job id given by the job manager. Used as a foreign key for Jobs.
- step_id: step id given by the job manager. Used as a foreign key for Jobs.
- value: value for the event. The units and semantic depend on the type of event. node_id: the node in which the application ran. The names of the nodes are trimmed at any ".", i.e., node1.at.cluster becomes node1.

  The origins of an event are indicated by its cardinality:

  1. EARL events' type is always $<$ 100
  2. EARD init events' type is always $>$=100 $<$=200
  3. EARD runtime events' type is always $>$=300 and $<$=400
  4. EARD powercap events' type is always $>$=500 and $<$=600
  5. EARGM events' type is always $>$=600 and $<$=700

  Certain events do not require a value, so it is set to 0 by default on those cases.

16.6.8 Global_energy

This table is used by the EARGM.

- energy_percent: percentage of consumed energy from the current budget.
- warning_level: current level of closeness to the current energy budget. Higher level means closer to the current budget.
- time: timestamp of the energy event
- inc_th: threshold increment sent to the EARDs to be applied to policies
- p_state: p_state variation sent to the EARDs
- GlobEnergyConsumedT1: current energy consumed within the last period T1
- GlobEnergyConsumedT2: current energy consumed within the last period T2
- GlobEnergyLimit: current energy budget/limit
- GlobEnergyPeriodT1: duration of the current period T1
- GlobEnergyPeriodT2: duration of the current period T2
- GlobEnergyPolicy: current energy policy used by the EARGM

  The warning level also indicates which inc_th and p_states are being sent to the EARDs

16.6.9 Periodic_metrics

- id: unique id generated by the database engine to use as primary key.
- start_time: timestamp of the start of the period
- end_time: timestamp of the end of the period
- DC_energy: total energy consumed by the node during the period in Joules
- node_id: the nodename in which the application period was registered. The names of the nodes are trimmed at any ".", i.e., node1.at.cluster becomes node1.
- job_id: job id given by the scheduler. Used as a foreign key for Jobs. If no job is running in the node during the period it will be 0.
- step_id: step id given by the scheduler. Used as a foreign key for Jobs. If no job is running in the node during the period it will be 0.
- avg_f: average CPU frequency (includes all the cores of the node) in Khz during the period.
- temp: average temperature reported by the node during the period.
- DRAM_energy: total energy consumed by the DRAM (includes 2 sockets) during the period, in Joules
- PCK_energy: total energy consumed by the CPU (includes 2 sockets) during the period, in Joules
- GPU_energy: total energy consumed by the GPU (includes all GPUs) during the period, in Joules

16.6.10 Periodic_aggregations

- id: unique id generated by the database engine to use as primary key
- start_time: timestamp of the start of the period
- end_time: timestamp of the end of the period
- DC_energy: accumulated energy consumed by the period
- eardbd_host: hostname of the eardbd reporting the data to database. The hostnames of the nodes are trimmed at any ".", i.e., service1.at.cluster becomes service1.

Chapter 17

Energy Data Center Monitor

The Energy Data Center Monitor is a new EAR service for Data Center monitoring. In particular, it targets elements different than computational nodes which are already monitored by the EARD running in compute nodes. However, whereas the EARDs monitor (among others) DC node power, the EDCMON service targets (eventhough it's not limited to) AC power. Because of that reason, the EDCMON main goal is to include all the power consumer components in a Data Center (Compute nodes, Network, Storage, Management).

EDCMON is 100% configurable and extensible since it uses an EAR framework named Plugin Manager which allows to load as many plugins as needed, which specific frequencies , dependencies among them and to share data between them. These plugins can communicate with each other through a **tag** (naming) system. The tag is a free text specified in the plugin code and is used as reference to specify dependencies, data sharing etc.

17.1 The EDCMON executable

EDCMON parameters are:

```
Usage: ./edcmon [OPTIONS]

Options:
    --plugins    List of comma separated plugins to load.
    --paths      List of comma separated priority paths to search plugins.
    --verbose    Show how the things are going internally.
    --silence    Hide messages returned by plugins.
    --monitor    Period at which the plugin wake ups for monitoring. Def=100 ms
    --relax      Period to be used during low monitoring periods. Def=100 ms
    --help       If you see it you already typed --help.
```

This is an example of the executable arguments and its format:

```
edcmon --monitor=1000 --relax=1000
      --plugins=nodesensors.so:30000+nodesensors_report.so:30000:nodesensor_log+nodesensors_alerts.so:30000:log
      --verbose
```

This example shows the default configuration used by EAR when the edcmon service is deployed. This case configures a monitoring period of 1 second and it loads three plugins (separated by character +):

- nodesensors: monitoring plugin based on Lenovo Confluent software. Reads specified sensors every 30 seconds. Exposes "nodesensors" tag.
- nodesensors_report: a reporting plugin for nodesensors plugin. Depends on "nodesensors" tag and uses its data. It is executed every 30 secs. The "nodesensor_log" is a parameter indicating the report plugin to use. Exposes "nodesensors_report" tag.
- nodesensors_alerts: An alerting plugin depending on "nodesensors" tag and using the data produced by it. Executed every 30 secs. Exposes the "nodesensors_alerts" tag. The argument "log" indicates the approach to report alerts.

Plugins are installed in $EAR_INSTALL_PATH/lib/plugins/monitoring folder

```
./edcemon --plugins=metrics.so:2000+periodic_metrics.so:4000 --paths=path/to/plugins1:path/to/plugins2
```

The list of plugins to load contains also their calling time in milliseconds. Its main periodic action (PA) function will be called once that time has passed. But that variable is not mandatory, because some plugins may not have defined a PA function and only act as receiver of other plugin data. In that case these receiving functions will be called once the shared data of other plugin is ready. Or maybe you don't want Plugin Manager to call your PA function in that moment.

Also, additional colons can be provided to pass information to a plugin during its initialization:

```
./edcmon --plugins=metrics.so:2000+periodic_metrics.so:4000:config_message1:config_message2
      --paths=path/to/plugins1:path/to/plugins2
```

You can send N configuration messages to your plugin initialization function which will alter its behaviour. You can avoid the time variable or write 0 instead:

```
./edcmon --plugins=metrics.so:2000+periodic_metrics.so:0000:config_message1:config_message2
      --paths=path/to/plugins1:path/to/plugins2

./edcmon --plugins=metrics.so:2000+periodic_metrics.so:config_message1:config_message2
      --paths=path/to/plugins1:path/to/plugins2
```

Plugins also have **dependencies**. It means that a plugin may depend on the actions or data shared by other plugins. A dependency is written in a string in the compiled binary itself, so you don't have to load it manually. It will be loaded automatically and its calling time could be the dependent plugin time (if specified in the binary). But if you want to set a specific calling time you have to load it manually and set the time you want. If a dependency is hard (which is specified in the string), a failure in the required plugin will disable the dependent plugin.

Plugins also have **priorities**. If a plugin A is a dependency of plugin B, the plugin A will be called before. If a Plugin B was written before plugin A in the `--plugins` parameter, A will be called before, because these cases are contemplated in the dependency system algorithmics.

17.2 EDCMON plugins

Even though in the plugins folder there are other plugins available (listed at the end of this page), these are the plugins specifics for Data center monitoring.

Plugin	Information
nodesensors	Reads confluent power sensors
nodesensors_report	Reports power readings explosed by nodesesors
nodesensors_alter	Checks limits and executes actions based on nodesensors

17.3 Creating new plugins

As previously said, the plugin periodic functions have to have concrete name. These functions names and arguments are the following:

```
void  up_get_tag         (cchar **tag, cchar **tags_deps)
char *up_action_init     (cchar *tag, void **data_alloc, void *data)
char *up_action_periodic (cchar *tag, void *data)
char *up_post_data       (cchar *msg, void *data)
```

The function `up_get_tag` is in charge of returning the plugin own tag and its dependency tags. A tag matches the name of the shared object file (without the extension). As seid, the dependency tags allows the Plugin Manager to search and open the tagged plugins automatically. The format is a tag list sepparated by plus signs. Example:

```
void up_get_tag(cchar **tag, cchar **tags_deps)
{
    *tag = "some_test";
    *tags_deps = "dependency1+!dependency2";
```

```
}
```

If a dependency tag starts with some symbols such as exclamation mark '!', it means that dependency is mandatory for the loading plugin, and in case it is not resolved the loading plugin will be disabled. The symbol '<' tells the Plugin Manager to inherit the timing of the dependant plugin.

The function `up_action_periodic()` or PA is the core function to perform actions and share data. It receives a tag and a pointer to the data associated with that tag. The received tag could be the self tag or the tag of other plugins. The plugin PA function will be called with its own tag and data when the specified time in `--plugins` argument has passed, or with other plugin tag and data after that plugin has called its own PA function with its own tag.

Examples of PA function types:

```
char *up_action_periodic(cchar *tag, void *data)
{
    if (is_tag("tag2")) {
        type2_t *d = (type2_t *) data;
        // work
    }
    return NULL;
}

char *up_action_periodic_tag1(cchar *tag, void *data)
{
    type1_t *d = (type1_t *) data;
    // work
    return NULL;
}
```

As you can see, you can define a generic `up_action_periodic()` function or one with a suffixed tag. A suffixed function will be called only when a plugin whose tag matches the function tag suffix. If you define just a generic version of the function, take into the account that you have to distinguish between tags. The macro `is_tag` will help you to do this and maintain your code clean and verbose.

The returning char is a message that Plugin Manager will print in case is not NULL. You can add some modifiers at the beginning of the message:

- `[D]` disables the plugin. It also re-activates the dependency system and could disable dependant plugins.
- `[=]` pauses the periodic call.
- `[X]` closes the Plugin Manager main thread.

The `up_action_init()` function works the same, it can receive the own plugin tag o other plugin tag. It is called one time before calling any PA function and can be used to allocate and initialize data.

```
static mydata_t mydata;

char *up_action_periodic_mytag (cchar *tag, void **data_alloc, void *data)
{
    *data_alloc = &mydata;
    return "I have been initialized and mydata will be shared among the loaded plugins";
}

char *up_action_periodic_tag2 (cchar *tag, void **data_alloc, void *data)
{
    tag2_type_t var = (tag2_type_t) data;
    return "Now I know that tag2 plugin has been initialized";
}
```

When an initialization function is called and receives its own plugin tag, the `data_alloc` double pointer serves as pointer to the data that self plugin wants to share with other plugins, so it is responsible to allocate the data and set the address pointer. When an initialization function is called and receives other plugin tag, the `data_alloc` variable is NULL and `data` parameter points to the shared data newly initialized by their own plugin, which is the same data referenced in the PA function.

The **configuration** string mentioned in EDCMON executable is also received when the initialization function is called with own plugin tag using the `data` parameter, and can be retrieved as a list of arguments:

```
static mydata_t mydata;

char *up_action_periodic_mytag (cchar *tag, void **data_alloc, void *data)
{
    char **args = (char **) data;
    *data_alloc = &mydata;
    if (args != NULL) {
        if (strcmp(args[0], "i_want_to_say_hello") == 0) {
            printf("Hello!\n");
        }
    }
    return "I have been initialized and mydata will be shared among the loaded plugins";
}
```

The final pipeline is:

```
1) up_get_tag (all plugins)
2) up_action_init (all plugins)
3) up_action_periodic (all_plugins)
4) up_action_periodic (the plugins whose time has passed, and then the plugins which depends on their data)

PA example, if B and C depends on A in --plugins=A.so:4000+B.so:3000+C.so:4000
 1) A up_action_periodic will be called with 'A' tag.
 2) B up_action_periodic will be called with 'A' tag.
 3) C up_action_periodic will be called with 'A' tag.
 4) B up_action_periodic will be called with 'B' tag.
 5) C up_action_periodic will be called with 'C' tag.
 6) After 3 seconds, B up_action_periodic will be called with 'B' tag.
 7) After 4 seconds, A up_action_periodic will be called with 'A' tag.
 8) After 4 seconds, B up_action_periodic will be called with 'A' tag.
 9) After 4 seconds, C up_action_periodic will be called with 'A' tag.
10) After 6 seconds, B up_action_periodic will be called with 'B' tag.
11) After 8 seconds, A up_action_periodic will be called with 'A' tag.
12) And so on...
```

Finally, `up_post_data()` allows to receive external data to the plugins. In example, if Plugin Manager is being used by the EARD, when a job starts you could send a message to the framework containing the job and step IDs. By now you can distinguish the messages by `is_msg` macro. Maybe in the near future we implement the suffix system too.

17.3.1 Helper macros

The following helper macros to define the functions and maintain your plugins updated in case of a change in some function. They can be found in plugin_manager.h:

```
#define declr_up_get_tag()               void   up_get_tag                 (cchar **tag, cchar **tags_deps)
#define declr_up_action_init(suffix)     char * up_action_init##suffix     (cchar *tag, void **data_alloc,
      void *data)
#define declr_up_action_periodic(suffix) char * up_action_periodic##suffix (cchar *tag, void *data)
#define declr_up_post_data()             char * up_post_data               (cchar *msg, void *data)
```

An example of the `up_action_periodic()`: Examples of action_periodic function types:

```
declr_up_action_periodic(_tag1)
{
    type1_t *d = (type1_t *) data;
    // work
    return NULL;
}

declr_up_action_periodic()
{
    if (is_tag("tag2")) {
        type2_t *d = (type2_t *) data;
        // work
    }
    return NULL;
}
```

17.4 Plugin Manager functions

By now these are the functions of the framework:

```
// Init as main binary function
```

```
int plugin_manager_main(int argc, char *argv[]);

// Init as a component of a binary
int plugin_manager_init(char *files, char *paths);

// Closes Plugin Manager main thread.
void plugin_manager_close();

// Wait until Plugin Manager exits.
void plugin_manager_wait();

// Asking for an action. Intended to be called from plugins.
void *plugin_manager_action(cchar *tag);

// Passing data to plugins. Intended to be called outside PM.
void plugin_mananger_post(cchar *msg, void *data);
```

The `plugin_manager_main()` receives the program arguments (argc and argv), in which is included `--plugins`. You can also call `plugin_manager_init()` if you prefer the list of plugins and search paths separately (but in the same format). `plugin_manager_action()` calls the PA function of the plugin whose tag is referenced, it can be useful in some contexts when a plugin prefers to call a required plugin manually. Finally `plugin_manager_wait()` waits until the Plugin Manager main thread is closed.

17.4.1 Other plugins already available

Plugin	Information
conf	Reads ear.conf and shares its data with other plugins.
dummy	Just an example.
eardcon	Connects with EARD. Saves other plugins to do that.
kernel_cl	An OpenCL kernel test.
kernel_cuda	A CUDA kernel test.
keyboard	A keyboard input.
management	Initializes all management APIs.
management_viewer	Views all management information.
metrics	Initializes and read all metrics APIs
metrics_viewer	Views all metrics readings.
periodic_metrics	Receives metrics and computes a periodic_metric.
test_cpufreq	A CPUFreq test.
test_gpu	Initializes and read the GPU API.

17.5 FAQ

- **Can I load the same plugin twice?** No.
- **Is the tag mandatory value?** Yes, all the plugins require a tag.
- **And the dependency tags?** Can be set to NULL if the plugin does not have any dependency.
- **Do I have to specify the time of a plugin in the dependency list?** No, is not recommended. A plugin which is loaded by the dependency list instead using the `--plugins` parameter inherits the dependent plugin time if using the special character '<' at the beginning of the string.
- **If none of the dependencies are resolved, the plugin periodic function will be called anyways?** Depends if some of the dependencies are mandatory, specified by the exclamation mark (!).
- **What happens if a plugin has periodic time specified but haven't defined a periodic function?** If there is no periodic function, nothing will be called.

- **Do I have to define all the API functions in the plugin?** No, only those necessary for the correct plugin functionality. The `get_tag` function is the exception because the tag is a mandatory value.
- **Can `action_init` function be defined but `action_periodic` not?** Yes. Sometimes you want to perform an action just once and you can do it in the init function. In example, the job of the plugin `conf.so` is to read `ear.conf` and pass the configuration structure to the rest of loading plugins.
- **For a plugin which does not allocate data, is its periodic function called?** Yes, if it's defined. But the NULL value in the allocated data pointer disallows any information exchange, so periodic function of other plugins wont be called.
- **If a plugin has defined the a function `action_periodic_tagX` for the tag `tagX`, but also the general `action_periodic`, which of the two would be called?** If defined a suffixed function, that tagged version will be called. For the rest of the tags the general `action_periodic`.

Chapter 18

Changelog

18.1 EAR 5.1

- CPU temperature read by EARL and reported to csv files.
- Prevent workflows where all applications see all GPUs and all of them change GPU frequency.
- Support for Python applications which use multiprocess module.
- EARL is compiled by default with LITE mode and DCGM metric collection enabled by default.
- Fix DCGM application-level metrics computation.
- Prevent closing fd 0 on NTASK_WORKSHARING use case.
- Avoid collapsing application's channel 2 when earl has high verbosity.
- Checking for authorized groups fixed.
- Added a tool for creating application-level signatures csv file from loop signatures csv file.
- EARL Loader can detect Python MPI flavour without an environment variable.
- Fixed an error with EARD remote connections not being properly closed.
- Add –domain option to econtrol.

18.2 EAR 5.0.3

- EARD local API creates an application directory if a third-party program connects with it.
- Fixed a typo in ereport queries.
- Prevent closing fd 0 on NTASK_WORKSHARING use cases.
- Prevent closing fd 0 when initiating earl_node_mgr_info.

18.3 EAR 5.0

- Workflows support. Automatic detection of applications executed with same jobid/stepid.
- Fixed Intel PSTATE driver to avoid loading if there is a driver already loaded.
- Robustness improved.
- OneAPI support for Intel PVC GPUs.
- EAR Data Center Monitoring component added.
- Improved metrics and management APIs.
- Detect the interactive step on slurm systems >= 20.11 if LaunchParameters contains use_interactive_step in slurm.conf.
- Support for getting NVIDIA DCGM metrics.
- enode_info tool added.
- Process resource usage is now reported by the EAR Library logs.
- Support for non-MPI multi-task applications, when tasks are spawned at invokation time, not from the application itself.
- Fixes in EAR Loader to support MPI application when MPI symbols can not be detected.
- GPU GFLOPS are now estimated and reported when using NVIDIA GPUs.

18.4 EAR 4.3.1

- Documentation typos fixed.
- EAR configuration files templates updated.
- Bugs fixed for intel_pstate CPUFreq driver support.
- Powercap bug fixes.
- ear.conf parsing errors found and fixed.

18.5 EAR 4.3

- MPI stats collection now is guided by sampling to minimize the overhead.
- EARL-EARD communication optimized.
- EARL: Periodic actions optimization.
- EARL: Reduce time consumption of loop signature computation.
- erun: Provide support for multiple batch schedulers.
- eardbd: Verbosity quality improved.
- Improved metrics computation in AMD Zen2/Zen3.
- Improved robustness in metrics computation to support hardware failures.

18.6 EAR 4.2

- Improved support for node sharing : save/restore configurations
- AMD(Zen3) CPUs
- Intel(r) SST support ondemand
- Improved Phases classification
- GPU idle optimization in all the application phases
- MPI load balance for energy optimization integrated on EAR policies
- On demand COUNTDOWN support for MPI calls energy optimization
- Energy savings estimates reported to the DB (available with eacct)
- Application phases reported to the DB (available with eacct)
- MPI statistics reports: CSV file with MPI statistics
- New Intel Node Manager powercap node plugin
- Improvements in the Meta-EARGM and node powercap
- Improvements in the Soft cluster powercap
- New report plugins for non-relational DB: EXAMON, Cassandra, DCDB
- Improvements in the ear.conf parsing
- Improved metrics and management API
- Changes in the environment variables have been done for homogeneity

18.7 EAR 4.1.1

- Select replaced by poll to support bigger nodes
- Minor changes in edb_create and FP exceptions fixes

18.8 EAR 4.1

- Meta EARGM.
- Support for N jobs in a node.
- CPU power models for N jobs.
- Python apps loaded automatically.
- Support for MPI-Python through environment variable.
- Report plug-ins in EARL, EARD and EARDBD.
- PostgreSQL support.
- Soft cluster powercap.
- New AMD virtual P-states support using max frequency and different P-states.

- New RPC system in EARL-EARD communication (including locks).
- Partial support for different schedulers (PBS).
- New task messages between EARPlug and EARD.
- New DCMI and INM-Freeipmi based energy plug-ins.
- IceLake support.
- Likwid support for IceLake memory bandwidth computation.
- msr_safe
- HEROES plug-in.

18.9 EAR 4.0

- AMD virtual p-states support and DF frequency management included
- AMD optimization based on min_energy and min_time
- GPU optimization in low GPU utilization phases
- Application phases IO/MPI/Computation detection included
- Node powercap and cluster powercap implemented: Intel CPU and NVIDIA GPUS tested. Meta EAR-GM not released
- IO, Percentage of MPI and Uncore frequency reported to DB and included in eacct
- econtrol extensions for EAR health-check

18.10 EAR 3.4

- Automatic loading of EAR library for MPI applications (already in 3.3), OpenMP, MKL and CUDA applications. Programming model detection is based on dynamic symbols so it could not work if symbols are statically included.
- AMD monitoring support.
- TAGS support included in policies.
- Request dynamic in eard_rapi.
- GPU monitoring support in EAR library for NVIDIA devices.
- Node powercap and cluster power cap under development.
- papi dependency removed.

18.11 EAR 3.3

- eacct loop signature reported.
- EAR loader included.
- GPU support migrated to nvml API.
- GPU support in configure.
- TAGS supported in ear.conf.
- Heterogeneous clusters specification supported.
- EARGM energy capping management improved.
- Internal messaging protocol improved.
- Average CPU frequency and Average IMC frequency computation improved.

18.12 EAR 3.2

- GPU monitoring based on nvidia-smi command.
- GPU power reported to the DB using NVIDIA commands.
- Postgresql support.
- freeipmi dependence removed.

Chapter 19

FAQs

19.1 EAR general questions

Q: What is EAR?

A: EAR is a system software for energy and power optimization, accounting and management. EAR components interact with each other to perform system monitoring, job accounting and energy optimisation on HPC systems.

The main goals of EAR are to provide energy optimization and monitoring about power consumption about the system or jobs.

Q: Must all EAR components be available to get EAR working?

A: No. Each set of components was designed to provide different services.

The minimal functionality is the system power monitoring, provided by the EAR Daemon (EARD). In order to get the information collected by the EARD, another service is needed to store such, i.e., the EAR Database Daemon (EARDBD).

In addition, EARD can offer job accounting if EAR is integrated with the system batch scheduler (e.g., SLURM). Current release has support for SLURM, PBSPro, OpenPBS and OAR. These integrations can load the EAR Library (EARL) when an application is starting in order to get performance metrics and apply energy optimisation policies.

Finally, the EAR Global Manager Daemon (EARGMD) is a cluster wide component offering cluster energy monitoring and capping. Its main goal is to provide features to apply power capping policies for systems with power consumption restrictions.

Q: Which programming models does EARL support?

A: EARL can be loaded automatically on pure MPI, MPI+OpenMP, OpenMP, MKL, CUDA, and python applications. For other programming models it must be loaded on demand.

Depending on the programming model, EARL offers different features. Also, the approach to load it may differ depending on which kind of application is launched. See the user guide to read more detailed information on how to run jobs with the EAR.

Q: Is EAR providing per-process metrics?

A: The current version reports per jobid/stepid/nodeid metrics, that is, metrics reported are or the accumulated or the averages of the metrics for all the processes of the application in the node. However, the EAR library collects per core and per-process metrics so it will be available in next versions.

Q: OK, so will I get per-thread metrics on Hybrid MPI+OpenMP apps?

A: Unfortunately no, you won't.

EAR does not provide metrics at thread level at this moment. For hybrid applications, EAR will provide metrics for each MPI process. You can run OpenMP apps with EARL, but you'll get process metrics too.

19.2 Using EAR flags with SLURM plug-in

Q: How to see EAR configuration and metrics at runtime?

A: Use `--ear-verbose=1` flag in your submission command (e.g., srun, mpirun) to enable verbosity.

Check the user guide to see how to tune EAR configuration at submission time.

Q: Why EAR flags are not working?

A: The following list of EAR flags are only allowed to Authorized users (contact with your system administrator):

- `--ear-cpufreq`
- `--ear-tag`
- `--ear-learning`
- `--ear-policy-th`
- `--ear-policy*`
- It depends on the system administrator whether this flag is restricted to authorized users. Check next question.

Below there are `ear.conf` fields that specify the list of authorized users/accounts/groups:

```
AuthorizedUsers=user1,user2
AuthorizedAccounts=acc1,acc2,acc3
AuthorizedGroups=xx,yy
```

If a user is not authorized, non-working flags is the expected behaviour.

Q: Why is a different energy policy other than the selected one being applied? I validated it with `--ear-verbose=1`.

A: The selected policy may not be enabled for all users.

Energy policies can be configured to be enabled to all users or not. Contact with your system administrator.

Check policy configuration (ear.conf) and user authorization (ear.conf).

```
#Enabled to all users
Policy=monitoring Settings=0 DefaultFreq=2.4 Privileged=0
#Enabled to authorized users
Policy=monitoring Settings=0 DefaultFreq=2.4 Privileged=1
```

If not enabled or not authorized therefore this is is the expected behaviour.

Q: How to disable the EAR library explicitly?

A: Use `-ear=off` flag at submission time.

Q: Can I apply EAR settings to all runs (e.g., srun, mpirun) inside a batch script?

A: Yes, you can provide EAR flags to the batch scheduler integration by setting EAR options on the header of your batch script. For example, in SLURM systems:

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -ear-policy=min_time # application 1 and 2 will run with min_time

srun application1

srun --ear-verbose=1 application2 # This step will show EAR messages.
```

Q: How can I know which energy policies are installed?

A: Type `srun --help`. The output will show which policies you can provide with the flag `--ear-policy`.

It is possible that non-authorized users are not allowed to select EAR policy. Contact with your system administrator.

Q: How to set EAR flags with `mpirun` if I run my application with Intel MPI?

A: Depending on the Intel MPI version. Check the user-guide.

Before version 2019, `mpirun` had 2 parameters to specify SLURM options.
```
mpirun -bootstrap=slurm -bootstrap-exec-args="--ear-verbose=1"
```

Since version 2019, SLURM options must be specified using environment variables:
```
export I_MPI_HYDRA_BOOTSTRAP=slurm
export I_MPI_HYDRA_BOOTSTRAP_EXEC_EXTRA_ARGS="--ear-verbose=1"
```

Q: How to set EAR flags with `mpirun` if I run my application with OpenMPI?

A: OpenMPI needs an extra support when `srun` is not used. EAR's **erun** command must be used.
```
mpirun erun -ear-policy=min_energy --program=application
```

Q: How to collect paraver traces?

A: Use the environment variables to enable the trace collection and to specify the path.
```
SLURM_EAR_TRACE_PLUGIN$EAR_INSTALL_PATH/lib/plugins/tracer/tracer_paraver.so
SLURM_EAR_TRACE_PATH=TRACES_PARAVER/
```

19.3 Using additional MPI profiling libraries/tools

EAR uses the `LD_PRELOAD` mechanism to be loaded and the PMPI API for a transparent loading. In order to be compatible with other profiling libraries EAR is not replacing the MPI symbols, it just calls the next symbol in the list. So it is compatible with other tools or profiling libraries. In case of conflict, the EARL can be disabled by setting `--ear=off` flag at submission time.

19.4 Jobs executed without the EAR Library: Basic Job accounting

For applications not executed with the EARL loaded (e.g., `srun` is not used or programming models or applications not loaded by default with EARL), EAR provides a default monitoring. In this case a subset of metrics will be reported to the Databse:

Metric	Unit
Accumulated DC energy	Joules
Accumulated DRAM energy	Joules
Accumulated CPU package energy	Joules
EDP	Energy–Delay Product
Maximum DC node power detected	Watts
Minimum DC node power detected	Watts
Execution time	Seconds
CPU average frequency	kHz*
CPU default frequency	kHz*

- The unit showed by `eacct` output is GHz.

DC node energy includes the CPU and GPU energy if there are. These metrics are reported per node, Job and Step IDs, so they can be seen per job and job and step when using `eacct` command.

19.5 Troubleshooting

User asks for application metrics with `eacct` command and no information appears in some of the columns in the output.

This means EARL was not loaded with the application or the application fails before the MPI_Finalize call, nor reporting application data.

After some time, user asks for an application metrics with eacct and application is not reported.

Try again after some minutes (applications are not reported immediately).

Chapter 20

Known issues

20.1 Python+MPI

The execution of Python+MPI+EAR is not 100% automatic. Given Python is not a compiled language, the EAR loader cannot detect the MPI version automatically. Check in the User guide how to deal with this use case.

20.2 Non-MPI applications compiled with MPI compilers

The EAR Loader has several methods for detecting the application use case to select the correct EAR Library to load. When it tests whether the application is MPI, if it finds `libmpi.so` symbols it will load an MPI version of the EAR Library. Therefore, the Library init method is called just after `MPI_Init` call function of the application, so, if your application is not calling this method, the EAR Library will not be started, and no metrics will be reported.

Index