EAR 5.2

Admin guide

EAR team

2025-10-23

Contents

Introduction

EAR 5.2 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 auto-

matically 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 easily 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 completely 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.

Admin guide

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 PostgresSQL. 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.
- **Scheduler plug-in:** A SLURM SPANK plug-in and a PBS Pro Hook which provide support for using EAR job accounting and loading EARL transparently for users.

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

Quick Installation Guide

This section provides summary of needed steps for compiling and installing EAR. The complete guide is left into another section, 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, 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.

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.

Target	Description
`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:

```bash
#!/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 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.

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
```

Deployment and validation

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.

Monitoring: EAR plugin

Slurm

- 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.

PBS

- Set up EAR PBS Hook (see the configuration section for more information).

> For a first test it is recommened to set **`default`**`=off` in the `ear_hook_conf.ini` to disable the automatic loading of the EAR library.

EAR scheduler plugins 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
```

In PBS, to see EAR options run `ear-hook-help`. You must see something like below as part of the output. The EAR must be loaded.

For PBS:

```
[user@hostname ~]$ module load ear
[user@hostname ~]$ ear-hook-help
```

- Submit one application via the scheduler 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 `sbatch --ear=on` or `qsub -v "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` or in `hook_config.ini`.

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.

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.

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.

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

Library	Minimum version	References
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

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 `GlobalmanagerUseLog` 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`.

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`.

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.

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.

Architectures

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.

GPUs

NVIDIA

- From Turing to Hopper.

Intel

- PVC.

Operating systems

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

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.

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.

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.

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**.

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).

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.

AMD system management features

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

Performance counters

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

Learning phase

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

Installation from source

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

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.

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.

Variable	Description
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'.
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.

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.

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.

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.

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.

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.

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.

Architecture

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.

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.

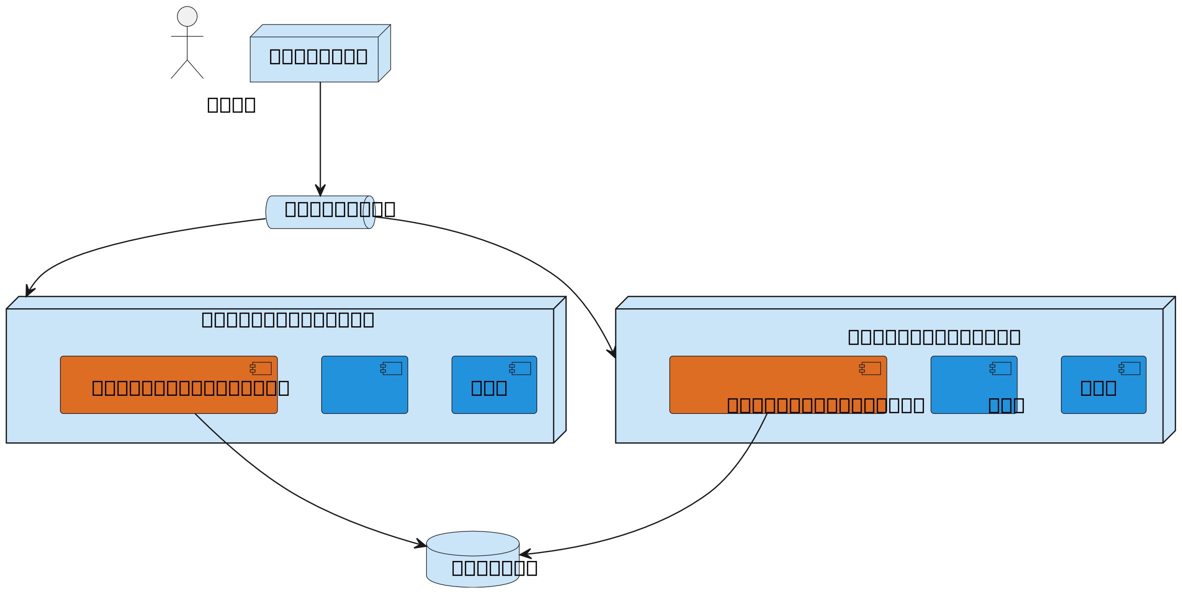


Figure 1: EAR_basic_accounting.svg

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.

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.

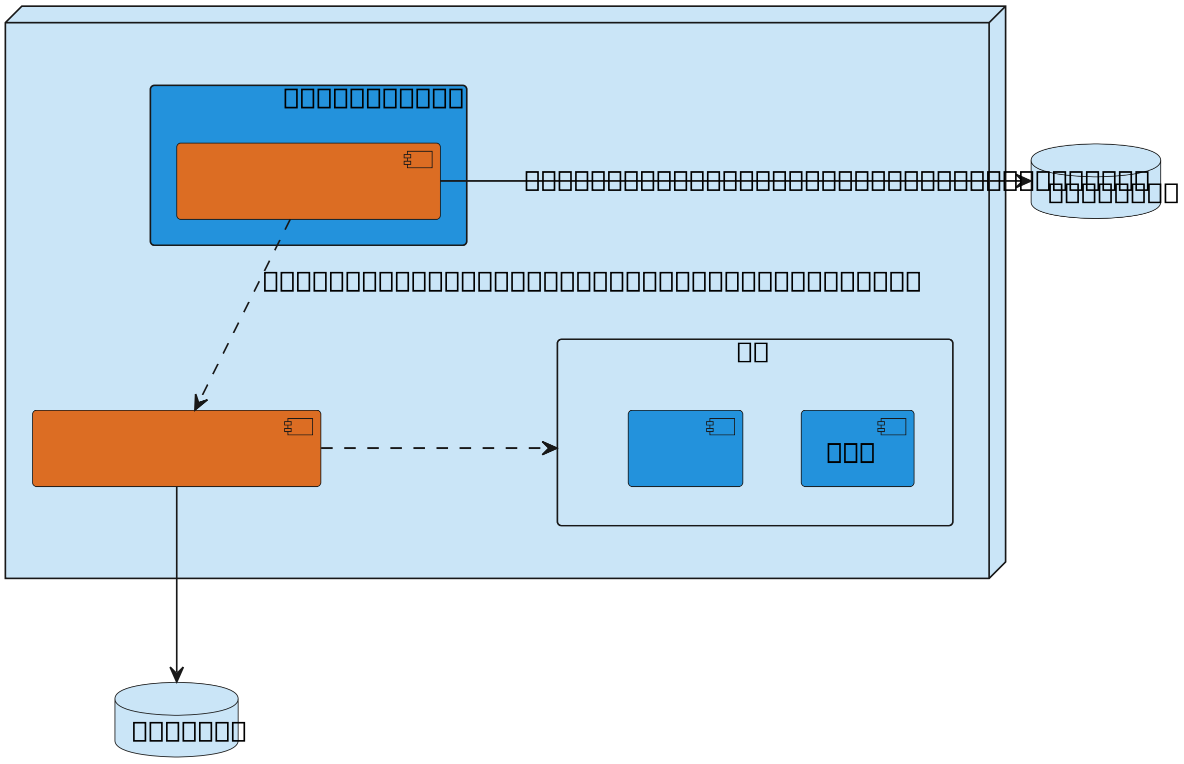


Figure 2: EAR_job_mgr.svg

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.

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.

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.

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.

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`

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.

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).

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.

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.

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.

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.

Local powercap EARGM has a local version that can be run without privileges and that controls the power consumption of a list of nodes. This can be used as a rudimentary version for job powercap, where a job with N nodes is not allowed to consume more than a certain amount of power. In the current version, if the allocated power is exceeded a powercap will be applied to all nodes equally (that is, the same amount of power will be allocated to each node, regardless of their actual consumption). Furthermore, custom scripts may be executed when the power reaches certain thresholds so the user can have more control over what to do.

See the execution section for how to run this mode.

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.

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.

To execute a local EARGM with powercap for certain nodes, one may run it as:

```
eargmd --powercap=2000 --nodes=node[0-4] --powercap-policy soft --
    suspend-perc 90 --suspend-action suspend_action.sh --powercap-period
    =10 --conf-path=$HOME/ear_install/etc/ear/ear.conf
```

This will execute an EARGM that will control nodes node[0-4], apply a total powercap of 2000W with a soft powercap policy (that is, the application will run as normal unless the aggregated power of all 5 nodes reaches 2000W, at which point a power limit of 400 per node will be applied).

`suspend-perc` indicates the percentage of power to reach for `suspend-action` to be executed; in this case, when power reaches 1800W suspend_action.sh will be called *once*. A reciprocal of this exists, called `resume-perc` and `resume-action` which will only be called once resume-perc power has been reached *AND* `suspend-action` has been called.

Finally, `powercap-period` sets the time between polls for power from the nodes (how often the EARGM checks the current power consumption), and `conf-path` specifies a custom ear.conf file.

For more information, one can run `eargmd --help`.

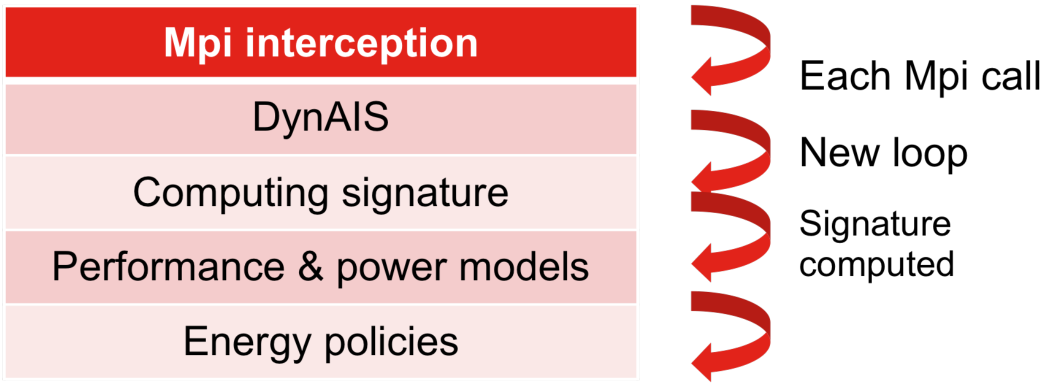
The EAR Library (Job Manager)

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.

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.

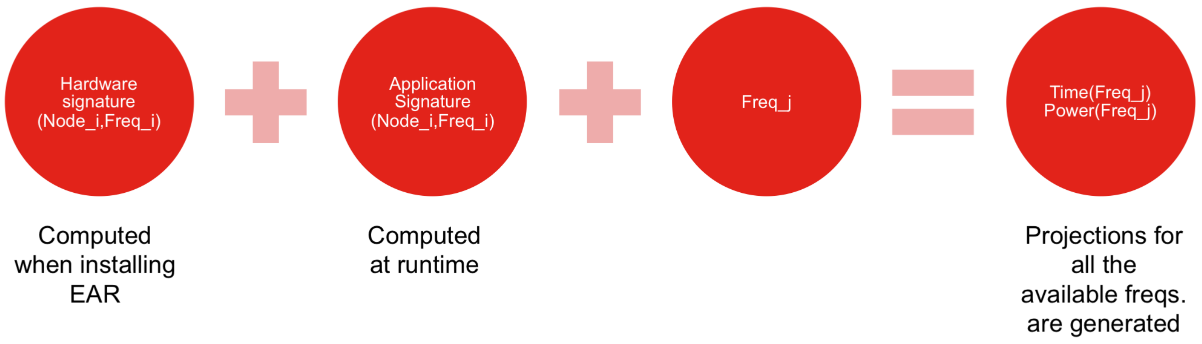
$$\mathrm{Power}(fn) = A(Rf,fn)*\mathrm{Power}(Rf) + B(Rf,fn)*\mathrm{TPI}(Rf) + C(Rf,fn)$$

$$\mathrm{CPI}(fn) = D(Rf,fn)*\mathrm{CPI}(Rf) + E(Rf)*\mathrm{TPI}(Rf) + F(Rf,fn)$$

$$\mathrm{TIME}(fn) = \mathrm{TIME}(Rf) * \mathrm{CPI}(Rf,fn)/\mathrm{CPI}(Rf) * (Rf/fn)$$

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.

3. **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.


Hardware signature (Node_i,Freq_i)
Application Signature (Node_i,Freq_i)
Freq_j
Time(Freq_j) Power(Freq_j)
Computed when installing EAR
Computed at runtime
Projections for all the available freqs. are generated

4. **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.

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.

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.

Classification

In the context of the Library's pipeline, phase classification is the module that, given the last computed application signature, undertakes the task of identifying the type of activity of the application, thereby giving hints to optimize its execution. By approaching the application signature as a semantic expression of this activity, the classification allows for guiding (or even skipping, if possible) subsequent steps of the pipeline.

Taking this activity as what we call an *execution phase*, EAR currently accounts for the following types: - **Computational phases**, which focus on the activity of the CPU in a way that is useful for the Job Manager's pipeline. This type includes execution phases, such as: - CPU-bound: intensity of calculus-related operations, as measured by the Cycles per Instruction (or `CPI`) and Floating-point operations (or `GFLOPS`). - Memory-bound: intensity of accesses to (main) memory (as measured by `MEM_GBS` in GB/s), and the memory Transactions per Instruction (or `TPI`). - Mix: intensity distributed between both calculus-related operations, and accesses to main memory. - **Non-computational phases**, which focus on the activity of the CPU in a way that allows for applying pre-defined optimizations to the application. This type includes the following execution phases: - CPU busy wait: intensive usage of the CPU due to an active wait - IO-bound: intensive usage of input-output channels - MPI-bound: presence of lots of MPI calls

Due to the fact that identifying computational phases correctly can optimize the Library's pipeline, and given making solid distinctions between execution phases is a complex task, the classification strategy becomes a key element of the main action loop. Currently, EAR incorporates three different strategies: - **Default strategy**: EAR's default classification model is based on setting predefined ranges of values for `CPI` and `MEM_GBS` metrics. These ranges, defined according to the architecture's characteristics via expert knowledge, allow identifying the different execution phases on a fundamental level, and are available since EAR's installation. - **Roofline strategy**: this approach is based on the roofline model, which conducts bottleneck analysis of the architecture's peak floating point performance and memory traffic to characterize the activity of any application. This strategy becomes available once the peaks for both resources have been computed, and allows for identifying execution phase types in a simple and quick way in runtime. - **K-medoids strategy**: this approach is based on the classification offered by the k-medoids clustering method, originally derived from k-means. The strategy, which becomes available once EAR has enough data in the database, allows for a more flexible classification than that of previous strategies, while also allowing for regeneration over time, as needed.

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.

`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)
```

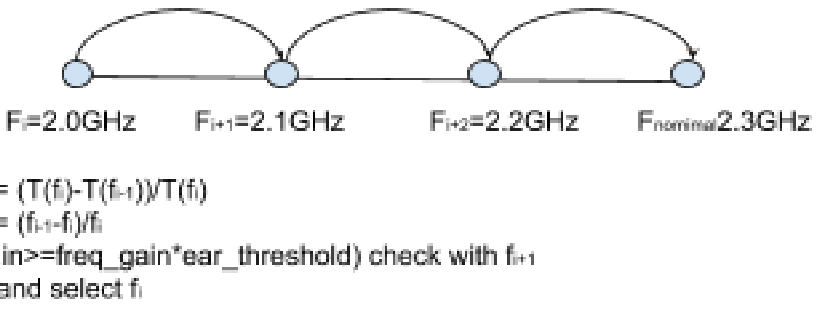
`min_time` The goal of this policy is to improve the execution time while guaranteeing a minimum ratio between performance benefit and frequency increment that justifies the increased energy consumption from this frequency increment. The policy uses the SLURM parameter option mentioned above as a minimum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



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

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

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.

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.

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.

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 you can read for more information.

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_ap 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();
}
```

High availability

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.

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.

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.

Configuration

EAR Configuration requirements

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

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.

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).

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/\\\*`.

EAR PBS Hook EAR supports PBS through the EAR PBSPro Hook.

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
```

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
```

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
```

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
```

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
```

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
```

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=
```

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
```

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
```

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
```

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
```

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`

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
```

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
```

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. For further information about EAR's database management tools, see the Commands section.

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

Next step

Visit the execution page to run EAR's different components.

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.

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>

Name	Description	Basic arguments
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.

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
```

EAR plugins

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.
Powercap	Powercap management.

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

> **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.

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.

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.

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.

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.

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.

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.

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
```

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

EARGM powercap mode	EARGM powercap value	Tag powercap value	Result
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

EARGM powercap mode	EARGM powercap value	Tag powercap value	Result
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.
```

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.

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.

Report plug-in name	Description
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.
dcgmi.so	Reports loop and application data to a CSV file. It differs from the csv_ts.so plugin since it also reports NVIDIA DCGM metrics collected by the EAR Library.

Prometheus report plugin

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.

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.

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.

```
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.

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.

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.

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.

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.

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.

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.

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

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/stepID/nodename/curr
- 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/stepID/nodename/avg/
- 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/stepID/nodename/curr

> 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

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
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
ACCOUNTID	This is the account of the user which ran the application. Only supported in SLURM systems.	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
ENERGY_TAG	The energy tag requested with the application (see `ear.conf`).	string
JOB_START_TIME	The timestamp of the beginning of the application, expressed in seconds since EPOCH.	integer
JOB_END_TIME	The timestamp of the application ending, expressed in seconds since EPOCH.	integer

Field	Description	Format
JOB_EARL_START_TIME	The timestamp of the beginning of the application monitored by the EARL, expressed in seconds since EPOCH.	integer
JOB_EARL_END_TIME	The timestamp of the application ending reported by the EARL, 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
POLICY	The Job Manager optimization policy executed (if applies).	string
POLICY_TH	The power policy threshold used (if applies).	real
JOB_NPROCS	The number of processes involved in the application.	integer
JOB_TYPE	The job type.	integer
JOB_DEF_FREQ	The default frequency at which the job started.	integer
EARL_ENABLED	Indicates whether the job-step ran with the EARL enabled.	integer
EAR_LEARNING	Whether the application was run in the learning phase.	
NODENAME	The short node name the following signature belongs to.	string
AVG_CPUFREQ_KHZ	The average CPU frequency across all CPUs used by the application, in kHz.	integer

Field	Description	Format
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
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

Field	Description	Format
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

Field	Description	Format
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

Field	Description	Format
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
NODEMGR_DC_NODE_POWER_W	Average node power along the time period, in Watts. This value differs from *DC_NODE_POWER_W* in that it is computed and reported by the Node Manager (the EARD) independently on whether the EARL was enabled.	real

Field	Description	Format
NODEMGR_DRAM_POWER_W	Average DRAM power along the time period, in Watts. **Not available on AMD sockets**. This value differs from *DRAM_POWER_W* in that it is computed and reported by the Node Manager (the EARD) independently on whether the EARL was enabled.	real
NODEMGR_PCK_POWER_W	Average RAPL package power along the time period, in Watts. This value shows the aggregated power of all sockets in a package. This value differs from *PCK_POWER_W* in that it is computed and reported by the Node Manager (the EARD) independently on whether the EARL was enabled.	real
NODEMGR_MAX_DC_POWER_W	The peak DC node power computed by the Node Manager.	real
NODEMGR_MIN_DC_POWER_W	The minimum DC node power computed by the Node Manager.	real
NODEMGR_TIME_SEC	Execution time period (in seconds) which comprises the job-step metrics reported by the Node Manager.	real
NODEMGR_AVG_CPUFREQ_KHZ	The average CPU frequency computed by the Node Manager during the job-step execution time.	real

Field	Description	Format
NODEMGR_DEF_FREQ_KHZ	The default frequency set by the Node Manager when the job-step began.	real

DCGMI

This plug-in reports same metrics as the CSV. Additionally, it reports NVIDIA DCGM profiling metrics for those NVIDIA GPU devices which support them.

> Since ear-v5.0, the EAR Library supports collecting and reporting NVIDIA DCGM profiling metrics for Ampere and Hopper devices. NVIDIA Turing should be supported as well.

Apart from loading the report plug-in, i.e., `export EAR_REPORT_ADD=dcgmi.so`, the EAR Library must have the DCGM monitoring enabled. This feature is enabled by default unless explicitely set at compile time. If disabled, you can enable it by setting the `EAR_GPU_DCGMI_ENABLED` environment variable to *1*:

```
...

export EAR_GPU_DCGMI_ENABLED=1
export EAR_REPORT_ADD=dcgmi.so
srun --ear=on my_app
```

Below table describes fields reported in the csv file generated by this plug-in. Please, review the official documentation for more information about each metric definition.

By default, **EAR just collects a subset of the DCGM metrics** (see below table). In order to collect all of them, set the `EAR_DCGM_ALL_EVENTS` environment variable to 1. See the full list of supported metrics:

Field	Description	Format
DCGMI_EVENTS_COUNT	The number of fields related with DCGM metrics.	integer
GPU*i*_gr_engine_active (*)	Graphics Engine Activity.	real
GPU*i*_sm_active (*)	SM Activity.	real
GPU*i*_sm_occupancy (*)	SM Occupancy.	real

Field	Description	Format
GPU*i*_tensor_active	Tensor Activity.	real
GPU*i*_dram_active	Memory BW Utilization.	real
GPU*i*_fp64_active	FP64 Engine Activity.	real
GPU*i*_fp32_active	FP32 Engine Activity.	real
GPU*i*_fp16_active	FP16 Engine Activity.	real
GPU*i*_pcie_tx_bytes (*)	PCIe Bandwidth (writes).	real
GPU*i*_pcie_rx_bytes (*)	PCIe Bandwidth (reads).	real
GPU*i*_nvlink_tx_bytes (*)	NVLink Bandwidth (writes).	real
GPU*i*_nvlink_rx_bytes (*)	NVLink Bandwidth (reads).	real

> * This metric needs to be requested explicitly through `export EAR_DCGM_ALL_EVENTS=1`.

EAR Database

Tables

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.

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).

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.

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.

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.

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

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.

Information reported and `ear.conf`

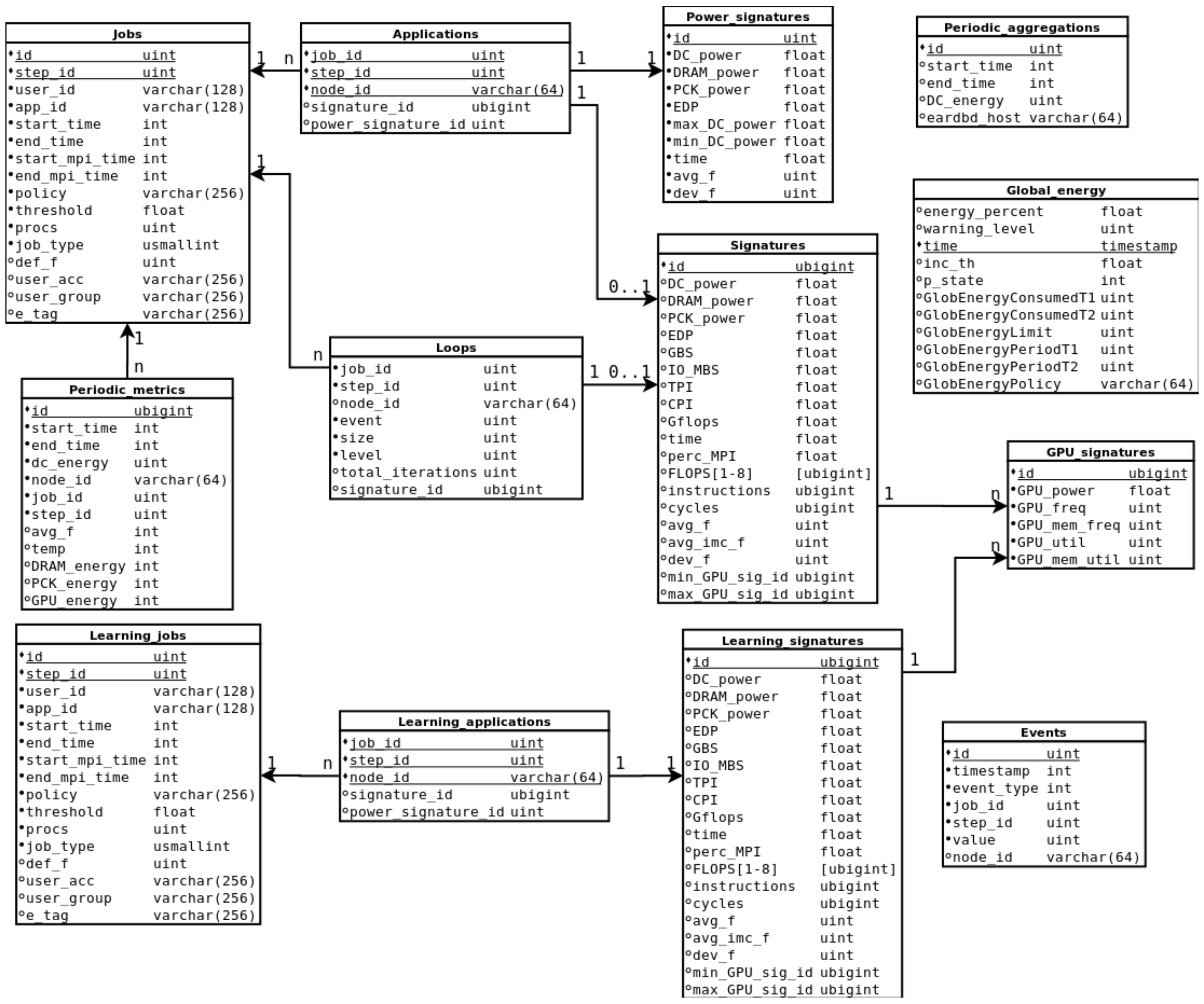
There are various settings in `ear.conf` that restrict data reported to the database and some errors might occur if the database configuration is different from EARDB's.

- `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.

Updating from previous versions

This section covers how to manually update the EAR Database tables if you update the EAR version and you want to maintain your current database data.

From EAR 5.0 to 6.0 Since 6.0 EAR reports the CPU utilization of applications in both loop and application signatures:

```sql
ALTER TABLE Signatures ADD COLUMN cpu_util INT UNSIGNED AFTER def_f;
```

From EAR 4.3 to 5.0 For better internal consistency, Jobs' id field was renamed:

```sql
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:

```sql
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);
```

```sql
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, node_id);
```

```sql
ALTER TABLE Loops ADD COLUMN local_id INT UNSIGNED NOT NULL AFTER
    step_id;
```

NOTE if the `id` in Jobs has been changed to `job_id`, the query to change primary keys will be:

```sql
ALTER TABLE Jobs DROP PRIMARY KEY, ADD PRIMARY KEY (job_id, step_id,
    local_id);
```

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).

```sql
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;
```

```sql
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;
```

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:

```sql
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:

```sql
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;
ALTER TABLE Periodic_metrics MODIFY COLUMN id BIGINT unsigned
    AUTO_INCREMENT;
```

If GPUs are being used, also run:

```sql
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;
```

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:

```sql
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;
```

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;
```

Database tables description

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

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.

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.

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.
> 2. Signatures are only reported when an application is running with EARL.
> 3. 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.

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

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.

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.
> 2. Loops will always have a signature because they are only reported when EAR is used
> 3. 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).

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.

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

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

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.

Architectures and schedulers supported

CPU Models

- Intel Haswell/Skylake/IceLake monitoring and optimization.
- AMD EPYC Rome/Milan monitoring and optimization.

GPU models

- NVIDIA: Node and application monitoring.

Schedulers

- EAR offers a SLURM SPANK plugin to be transparently used when using SLURM workload manager. This plug-in allows to be integrated as part of the SLURM submission options. See the user guide.
- PBS is supported through the EAR PBS Hook.
- Using the EARD api **new_job**/**end_job** functions EAR can be also be transparently used with other schedulers such as LSF through the prolog/epilog mechanism.

EDCMON

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.

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
```

Caution!! If the `config_message1` is a number, the Plugin Manager can understand that you are passing a time value instead an argument. You can set the calling time to 0 to avoid that misunderstanding.

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.

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

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

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's own tag and its dependency plugins tags. A tag must match the name of the shared object file **without the extension**, i.e., a plugin named `my_plug.so` has the tag *my_plug*. Thus, dependency tags allows the Plugin Manager to search and open the tagged plugins automatically. The format is a tag list separated by plus + signs. For example:

```
void up_get_tag(cchar **tag, cchar **tags_deps)
{
    *tag = "some_test";
    *tags_deps = "dependency1+!dependency2";
}
```

You can **prepend some symbols** to each dependency tag, which are used to set additional information about the dependency. This is the current set of dependency modifier symbols:

Symbol	Description
!	The dependency is mandatory.
<	Inherit the timing of the dependant plugin.

In the example above *dependency2* is set with the exclamation mark *"!"* prefixed, which means that the dependency is mandatory for the loading plugin, and in case it is not resolved the loading plugin will be disabled.

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:

```c
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's 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 initialise data.

```c
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 initialisation 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 initialisation function is called and receives other plugin tag, the `data_alloc` variable is NULL and `data` parameter points to the shared data newly initialised 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 initialisation 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.

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;
}
```

Plugin Manager functions

By now these are the functions of the framework:

```c
// 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.

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.

Plugin	Information
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.

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`.

CHANGELOG

EAR 5.1

- CPU temperature monitoring included in application monitoring and reported to csv files.
- Prevent workflows where all applications see all GPUs and all of them change GPU frequency.
- Support for Python multiprocess module.
- EARL is compiled by default with LITE mode. EAR Library will run in monitor mode if EARD is not present nor detected.
- Nvidia 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 stderr channel when EARL has high verbosity.
- Checking for authorized groups fixed. Support for multiple groups per user included.
- Added a tool for creating application-level signatures csv file from loop signatures csv file.
- EARL Loader extension for detecting Python MPI flavour more automatically.
- Fixed an error with EARD remote connections not being properly closed.
- Add –domain option to econtrol.

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.

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.

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.

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.

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

EAR4.1.1

- Select replaced by poll to support bigger nodes
- Minor changes in edb_create and FP exceptions fixes

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.

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

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.

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.

EAR 3.2

- GPU monitoring based on nvidia-smi command.

- GPU power reported to the DB using NVIDIA commands.
- Postgresql support.
- freeipmi dependence removed.