***A**ccurately **M**easuring **P**ower and **E**nergy for **H**eterogeneous **R**esource **E**nvironments*

University of Paderborn, Germany

Ampehre v0.5.10

Authors:
Achim Lösch
Christoph Knorr

September 8, 2015

Contents

1 Project Description

The Ampehre project is a BSD-licensed modular software framework used to sample various types of sensors embedded in integrated circuits or on circuit boards deployed to servers with a focus to heterogeneous computing. It enables accurate measurements of power, energy, temperature, and device utilization for computing resources such as CPUs (Central Processing Unit), GPUs (Graphics Processing Unit), FPGAs (Field Programmable Gate Array), and MICs (Many Integrated Core) as well as system-wide measuring via IPMI (Intelligent Platform Management Platform). For this, no dedicated measuring equipment such as DMMs (Digital Multimeter) is needed. We have implemented the software in a way that the influence of the measuring procedures running as a multi-threaded CPU task has a minimum impact to the overall CPU load. The modular design of the software facilitates the integration of new resources. Though it has been enabled to integrate new resources since version v0.5.1, the effort to do so is still quite high. Accordingly, our plans for the next releases are broader improvements on the resource integration as well as an extensive project review to stabilize the code base.

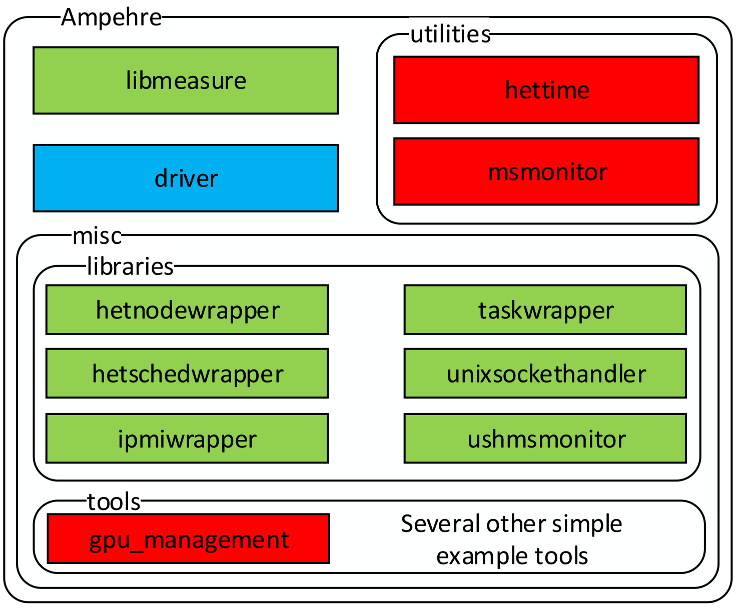

Figure 1: Ampehre project overview showing the directory structure (rounded rectangles) and different components (green: libraries, red: executables, blue: linux kernel module).

2 Component Overview

The Ampehre project consists of several libraries and executables to provide an easy extendable modular software framework to measure various physical values such as power, energy, and temperature of integrated circuits and boards of heterogeneous high-performance computers. Figure 1 presents an overview of project components and their location in the repository's directory structure. In the following paragraph we briefly describe each component.

libmeasure: Contains the main components of the ampehre project. The measuring library consists of several modules compiled to seperate shared objects used to measure physical values such as power, energy, temperature, and utilization of resources deployed to a single heterogeneous compute node. Each of these modules implements the measuring functionality related to one of the measured resources. We describe the structure of the modules in detail in section 3.

hettime: Allows us to measure the energy consumption, average power dissipation, maximum device temperature, and the CPU utilization of all supported resources while executing a binary given via command line. For this the utility uses the functionality provided by the libmeasure. The results are printed to the shell or stored as a csv.

msmonitor: Is a live monitoring tool which uses our libmeasure to retrieve current measurement values for power consumption, temperature, clock frequencies, utilization and the sum of allocated memory of the different resources. All values are shown in a Qt4-based user interface using the qwt library for plotting curves. This tool is also quite useful for debugging programms with unexpected "energy behavior".

driver: Contains a dynamically loadable kernel module. Our kernel module reads CPU MSRs (Model Specific Register), memory and swap occupancy, and the CPU utilization values provided by the Linux OS. Furthermore, the kernel module allows sending IPMI (Intelligent Platform Management Interface) requests to the BMC (Baseboard Management Controller). The driver functions are available through the character device `/dev/measure`.

ipmiwrapper: Provides functions to get the measured values of specific sensors via IPMI and also for DELL-specific IPMI requests. Internally, this library uses the `/dev/measure` device to send raw IPMI messages to the BMC and converts the response messages to double or integer values which are processed by the libmeasure.

gpu_management: Is used to set the GPU memory and core clock frequenices and to enable or disable the Nvidia driver's persistence mode. The NVML (Nvidia Management Library) is required by the tool and is usually installed alongside Nvidia driver and CUDA packages. You can find the corresponding header file in the Nvidia GPU Deployment Kit.

taskwrapper: Provides an interface to perform multiple energy measurements simultaneously. Since energy is the integral of power over time, we have to sample the power dissipation measured by device sensors periodically to compute a discrete approximation of the actual energy consumption. If multiple threads would perform different measurements, the same resources would be sampled concurrently, which has bad influence to the CPU load. The taskwrapper provides a simple mechanism to prevent concurrent measurements, by reading every sensor once every few milliseconds and calculating the energy consumed in the meantime individually for each registered thread. This feature can be useful, if you have one thread executing a GPU kernel while another thread is processing an FPGA kernel. This feature is less helpful, if you measure the energy consumption of multiple tasks running on the CPU simultaneously, as the energy consumption induced by one thread will invalidate the measurement result of the other task (and vice versa, of course).

hetnodewrapper: Provides a simplified interface to the libmeasure which allows multiple measurements. Only one measurement is instantiated in the libmeasure and the values of this measurement are sampled by the hetnodewrapper at the beginning and end of every measurement. The subtractions of the particular final and first measurements are the actual result used for returned measurements. The library retrieves runtime, energy, and utilization measurements for CPU, GPU, FPGA, and the mainboard respectively power supply.

hetschedwrapper: Uses the taskwrapper for measurements providing a more abstract interface. The wrapper is used by an external project not embedded in this repository.

unixsockethandler: Implements a communication layer for client-server IPC, based on Unix sockets with the advantage of a simple interface hiding the system calls.

ushmsmonitor: Is a library enhancing the general-purpose Unix socket-based IPC implementation of the unixsockethandler library. Our msmonitor monitoring utility uses the ushmsmonitor library as a server listening to a Unix socket. A client such as hettime must implement the counterpart. The clients can transmit start and stop signals to the server. Accordingly, servers are able to perform any activity after receiving these signals. For instance, out msmonitor tool shows start and stop markers on the qwt plots as well as the client's PID sending the corresponding signals.

Additionally, the `/misc/tools` directory contains several simple example programs which use the mentioned libraries. Each executable allows us to test a specific functionality of our project. **The `example_ms` tool shows all function calls which are necessary to perform measurements with our measuring library.**

3 Software Framework

In this section we first describe how to build and install our measuring framework. Secondly, we give a brief overview of libmeasure's modular software architecture. Finally, we explain what you have to do in order to add and implement a module to the library, so that you will be able to sample sensors provided by new devices.

3.1 Build and Install Ampehre

Ampehre is coded in C and C++. We use the cross-platform build system cmake to build the binaries and libraries. Additionally, we provide a GNU Makefile in the project's root directory that simplifies the build and install instructions. The difference between running make with the `Makefile` and a manual run of cmake is, that we build the software in a dedicated build directory and that the install directory is set to `/usr/ampehre`. This directory can be easily change by editing the `Makefile` and change the `BASE_DIR` variable. Do not forget to extend the environment variable `PATH` by `${BASE_DIR}/bin`. Furthermore, you must add `${BASE_DIR}/lib` to your system's `ld.so.conf`. It might be necessary to run `ldconfig` as root afterwards.

3.1.1 Prerequirements

This following software must be installed on your computer to build and run the software:

Software	Version
cmake	>= 2.6
GNU make	
gcc	>= 4.6
g++	>= 4.6
qwt	5.1.x
Qt4	>= 4.6

In order to compile our device-specific software components, you need the following packages installed. In order to build the CPU and IPMI modules, your system has to fulfill additional OS requirements.

Software	Version
CPU:	
cpufrequtils (libcpufreq)	
driver ⇒ Linux kernel and headers	2.6.32 *(CentOS 6.x)*
IPMI:	
Linux IPMI kernel module (ipmi_msghandler)	
driver ⇒ Linux kernel and headers	2.6.32 *(CentOS 6.x)*
GPU:	
NVML	>= 5.319 RC
FPGA:	
libmaxeleros.so	>= 2013.3
MIC:	
Intel MPSS (Manycore Platform Software Stack)	3.4.x
gaussblur:	
PGCC compiler	>= 14.7
correlation:	
nvcc compiler	>= 5.5

3.1.2 Build and Install Instructions

The easiest way to build and install the software is done by the shell commands as presented in the next listing. Note, that the user executing these programms must be allowed to be super user via `sudo`. This method installs the compiled project to `${BASE_DIR}/bin` and `${BASE_DIR}/lib`. If you want to develop your own additional software using one of our libraries you must call the functions provided as prototypes in our header files. You can find the headers in `${BASE_DIR}/include` after the installation is successfully completed. The header files installed to `${BASE_DIR}/include` are written in C, i.e., all public headers can be used in C as well as in C++ code.

```
$ make
$ make install
$ make driver
$ make driver_install
```

Listing 1: Default build instructions.

This will build and install the whole project but the gaussblur and correlation examples. Both subprojects are located in the `misc` directory and must be built seperately by the following commands:

```
$ make gaussblur
$ make gaussblur_install
$ make correlation
$ make correlation_install
```

Listing 2: Default build instructions for additional applications. Note, that you need the PGI OpenACC compiler as well as Nvidia's `nvcc`.

It is possible to build and install the project with a subset of supported resources. Therefore you can set the options shown in Listing 3 in the CMakeLists.txt of the root directory. This allows you to build and install the project if your system is not deployed with all resources for which we provide measuring functionality. Only the modules enabled in the CMakeLists.txt will be built. By default all options are on.

```
option(DELL_IDRAC7_SUPPORT
        "build library with dell idrac7 support" ON)
option(MIC_INTEL_KNC_SUPPORT
        "build library with mic intel knc support" ON)
option(GPU_NVIDIA_TESLA_SUPPORT
        "build library with gpu nvidia tesla support" ON)
option(FPGA_MAXELER_MAX3A_SUPPORT
        "build library with fpga maxeler max3a support" ON)
option(CPU_INTEL_SANDY_SUPPORT
        "build library with cpu intel xeon sandy support" ON)
```

Listing 3: CMakeLists.txt options.

3.1.3 Measuring library usage

The libmeasure can be called from other C or C++ projects by including the header file `/include/measurement.h` and linking against the `libms_common.so`. The complete user interface to the libmeasure is defined in the `measurement.h`. The first steps needed to perfom measurements are shown in the next Listing.

```
MS_VERSION version = { .major = MS_MAJOR_VERSION,
                       .minor = MS_MINOR_VERSION,
                       .revision = MS_REVISION_VERSION };
//init measuring library
MSYSTEM *ms = ms_init(&version, CPU_GOVERNOR_ONDEMAND,
                      2000000, 2500000, GPU_FREQUENCY_CUR);

// allocate measurement struct
MEASUREMENT *m = ms_alloc_measurement();

// Set timer for m. Measurements perform every 10ms/30ms.
ms_set_timer(m, CPU , 0, 10000000);
ms_set_timer(m, GPU , 0, 30000000);
ms_set_timer(m, FPGA , 0, 30000000);
ms_set_timer(m, SYSTEM , 0, 100000000);
ms_set_timer(m, MIC , 0, 30000000);
ms_init_measurement(ms, m, CPU | GPU | FPGA | SYSTEM | MIC);
```

Listing 4: Initialization of our measuring library `libmeasure`. Each function name has a `ms_` prefix.

At the beginning the libmeasure has to be initialized. Therefore the `ms_init` function is called. The first parameter is a `MS_VERSION` struct as defined in the `ms_version.h` header, to make sure that the correct version of the libmeasure is installed and used. The other parameters are used to set the CPU governor as available in the Linux kernel, CPU maximum and minimum frequencies and the GPU frequency. The next step is to allocate memory for the measurement results and all values related to a single measurement. This is done with a call to `ms_alloc_measurement` which returns a pointer to a so called `MEASUREMENT` struct. This struct contains variables for CPU, GPU, FPGA, MIC and System measurements such as the average power consumption of a resource, or the maximum temperature of a device retrieved during a concrete measuring. Before the measurement can be started, the sampling rate for each resource need to be set. This defines how often the actual measurements are executed and thus how often the current values are sampled from the resources. The sampling rates are set by the function `ms_set_timer`. The first parameter is the pointer to the `MEASUREMENT` struct which is used to hold the measured values as well as some temporary data. The second parameter is the resource for which the sampling rate is set. The third parameter is the number of seconds and the fourth parameter is the number of nanoseconds which are combined internally to a `struct timespec` to memorize the sampling rate. For further details on the sampling rates have a look at Appendix A. Note that you can set the sampling rates independently for each resource. This could be helpful, if you need precise measured data from a resource A, but only imprecise data from a resource B, while the total CPU load induced by the libmeasure should be limited somehow. The final step in the initialization phase is the call to `ms_init_measurement` which initializes the threads for the measurements. The last parameter specifies for which resources the measurement should be performed.

After the initialization the measurement can be started and stopped with the following functions. A measurement can be started and stopped exactly once, i.e., restarts to accumulate measurements are not supported yet. The `do_something()` function should be replaced by your specific code. While executing this code, the measuring library samples the sensors of the devices and stores energy, power, temperature, and so on in the `MEAUSREMENT` struct `m`. Our hettime tool replaces the `do_something()` function by forking a process and calling `execve()` with an executable given by the flag `-e` with the arguments of flag `-a` (Appendix B for further information).

```
ms_start_measurement(ms, m);

do_something();

ms_stop_measurement(ms, m);
```

Listing 5: The start and stop functions trigger the measuring procedures of the measuring library. Please replace the `do_something()` function by the code you want to execute while the measuring system is running.

Before the measured values can be retrieved the internal measurement threads need to be stopped and terminated. Therefore we call the functions shown in Listing 6. This function calls are necessary, since we internally use POSIX-Threads that would continue writing new data to the `MEASUREMENT` struct.

```
ms_join_measurement(ms, m);
ms_fini_measurement(ms, m);
```

Listing 6: Functions to join and terminate measurement threads.

Subsequently it is possible to retrieve the measured values. For each measured value a function is defined in the `measurement.h` to return the corresponding value. Listing 7 shows a few examples, a complete list of the available functions can be found in the `measurement.h`

```
printf("consumed energy of cpu 1 dram bank : %.2lf mWs\n",
        cpu_energy_total_dram(m, 1));
printf("maximum temperature of gpu         : %u \u00b0C\n",
        gpu_temp_max(m));
printf("total time of mic measuring        : %.2lf s\n",
        mic_time_total(m));
printf("mic temperature max die            : %u \u00b0C\n",
        mic_temp_max_die(m));
```

Listing 7: Example for getting the measured values.

Finally, you should free all memory allocated for the measurements and cleanup the environment. This is done by a call to the following functions.

```
ms_free_measurement(m);
ms_fini(ms);
```

Listing 8: Environment cleanup and freeing memory.

This frees the `MEASUREMENT` struct and thus deletes all measurement results.

3.2 Overview

Figure 2 shows the concept of the modular libmeasure software architecture with the two different types of libraries and their most important components. As shown in Figure 2, the `libms_common.so` is designed to be as independent as possible from the resource-specific implementations and therefore it contains abstract classes which are used for the management of all resource-specific modules. These resource-specific modules inherit from the abstract classes and contain the concrete implementations. In general there are multiple resource-specific modules which are controlled by the `libms_common.so`. For simplification, we always use the word *resource* as a template, that you can replace by any concrete resource. This is possible, since all resource-specific modules have the same structure. Below we briefly describe the most important classes.

`libms_common.so` contains resource-independent classes with some abstract methods which have to be implemented in inherited classes encapsulated in the resource-specific modules. Moreover, there are other classes which are common to all resource-specific modules. These classes are needed for the module management, library initialization, and so on.

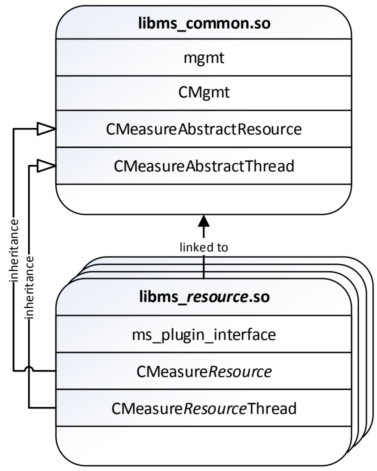

Figure 2: Design concept of the libmeasure software architecture. Each resource-specific module has to inherit from two abstract classes. Since the resource modules are dynamically loadable modules which can be loaded at runtime, they have an C interface to access the module (`ms_plugin_interface`).

- `CMeasureAbstractResource`: Abstract class for the measurement functionality of a resource. This class is used as skeleton for the concrete measurement class which is responsible to obtain the measurement values. Therefore this class provides a virtual `measure()` method which must be implemented by all resource-specific modules. Our measuring library calls the non-virtual `measure()` method of the inherited resource-specific class, if the time slot given by the resource's sampling rate is expired.
- `CMeasureAbstractThread`: Abstract class for the thread which periodically executes the `measure()` function for one resource in its inherited resource-specific classes. All thread management functions such as `start()`, `stop()`, or `join()` are implemented in this abstract class and must not be overwritten in the functions of the inherited classes. On the other side, the `run()` method of the thread is abstract and needs to be implemented in the concrete implementation in `libms_resource.so`. This class also enables controlling of the actual measurement threads, i.e., it initializes and stops the threads which trigger the resource-dependent `measure()` methods.
- `CMgmt`: This class is responsible for the management of all measurements. All POSIX-Threads executing the measuring procedures in the inherited implementations of `CMeasureAbstractThread` are instantiated in this class (in fact, the raw POSIX-Thread handling is performed in the `CMeasureAbstractThread` class and its children). In its constructor every module is dynamically loaded with `dlopen()`. This is still implemented statically. In the future, we are going to reimplement the module loader in a more dynamic way, so that modules are only loaded, if they are listed in a configuration file located in the user's home directory.

- `mgmt`: This class implements the libmeasure management functions such as `ms_init()`, `ms_start_measurement()` or `ms_join_measurement()` which can be called from C. We describe this functions in Section 3.1.3. These functions are accessible by including the corresponding interface header file `measurement.h`.

`libms_resource.so` encapsulates a resource-specific module with the concrete implementation of the abstract classes which are compiled to the shared object `libms_common.so`. As already mentioned, the word *resource* is used as a template and must be replaced by a concrete resource. We illustrate the structure of the resource-specific modules without considering the real class names but using the template word *resource*. Feel free to take a look to the source code for better understanding of the modules, respectively plugins.
Every resource with sensors that should be sampled by the libmeasure, must have a separate module with implementations of the abstract classes mentioned in the prior passage. Since the modules are loaded like plugins, there must be an implementation of the plugin interface, too.

- `CMeasureResource`: Implementation of the abstract class `CMeasureAbstractResource` and therefore also the `measure()` method. The resource-specific measuring functionality is implemented and all retrieved values are stored in the `MEASUREMENT` struct. No further calculations such as the integration of the retrieved power data to get the consumed energy since the last sample are done here. Such computations are done in the `run()` method of `CMeasureResourceThread`, i.e., the `run()` method calls the `measure()` method periodically, calculates additional data such as the consumed energy from raw power data stored in the `MEASUREMENT` struct, and finally stores these results in other elements of the `MEASUREMENT` struct. Furthermore, all resource-specific initialization should be done in the constructor of the `CMeasureResource` class (e.g. calling the external library NVML which is used for sampling sensors of Nvidia GPUs). Consequently, the destructor is used to close the libraries and/or free the memory allocated for using the libraries.

- `CMeasureResourceThread`: Concrete implementation of `CMeasureAbstractThread` and therefore the resource-specific `run()` method. This class uses the `CMeasureResource` class to retrieve measurement values periodically. The `run()` method basically consists of a loop frequently calling the `measure()` method of the `CMeasureResource` class. Afterwards all necessary calculations such as accumulation of energy values are done with the obtained values which are located in the `MEASUREMENT` struct. The `measure()` method only stores values which can be directly read from the sensors placed on-die or on-board of the resources. The further processing of the raw data is then performed in the loop of the `run()` method located in `CMeasureResourceThread` or after the loop before thread termination. For example, the recently measured power dissipation and time slice since the last sampling (approximately the sampling rate) are used to calculate the energy consumed during this period. Finally, this intermediate result is used to accumulate/sum up the total energy consumption during the whole measurement respectively between calling `ms_start_measurement(ms, m)` and `ms_stop_measurement(ms, m)` (Section 3.1.3). The calculated values are stored in the `MEASUREMENT` struct again.

- `ms_plugin_interface`: This is the C interface of the module respectively plugin which is called from the `CMgmt` class in order to instantiate the `CMeasureResource` and `CMeasureResourceThread` objects. Since the interface is written in C but the library encapsulates C++ classes, the returned void pointers are usually related to objects.

Figure 3 illustrates in which way the classes interact in order to perform the measurements. Moreover, there is a rough overview of the topmost functions users have to call in order to use libmeasure (functions with `ms_` prefix). For simplification only one resource is shown. Therefore, the keyword *Resource* is used again.

On the one side, all resource-specific libraries have to be linked against the `libms_common.so`. On the other side, `libms_common.so` does not need to be linked against the resource-specific implementations. This is not necessary, because the `CMgmt` class *dynamically loads the modules at runtime like plugins*, calls the interface, obtains the module-internal objects as void pointers, and stores the objects in their abstract types. Since we dynamically load the resource-specific shared objects at runtime, we have to compile the modules as dynamically loadable modules. Take a look to the `CMakeLists.txt` files to see how to compile appropriately.

Figure 4 shows the libmeasure with all currently available modules. There you can also see the replacement of *resource* by real class names. All modules have the same structure and implementations of the abstract classes from the `libms_common.so`. Caused by some "historic" reasons, our naming scheme is inconsistent. For instance, instead of naming the `CMeasureResource` and `CMeasureResourceThread` classes in `libms_gpu_nvidia_tesla_kepler.so` `CMeasureGPU` and `CMeasureGPUThread`, we used the name of the library (NVML) that we utilize to retrieve the measurement values.

As we already mentioned in Section 3.1.2, it is possible to build and install the project with a subset of supported resources. In this case the module `libms_stub.so` is loaded instead of the disabled resource-specific modules. The stub module has no functionality but defines all necessary classes and functions. This allows loading the stub module in the `CMgmt` class like all other modules. If the functions are called, no measurement thread is created and the values stored to the `MEASUREMENT` struct are always zero.

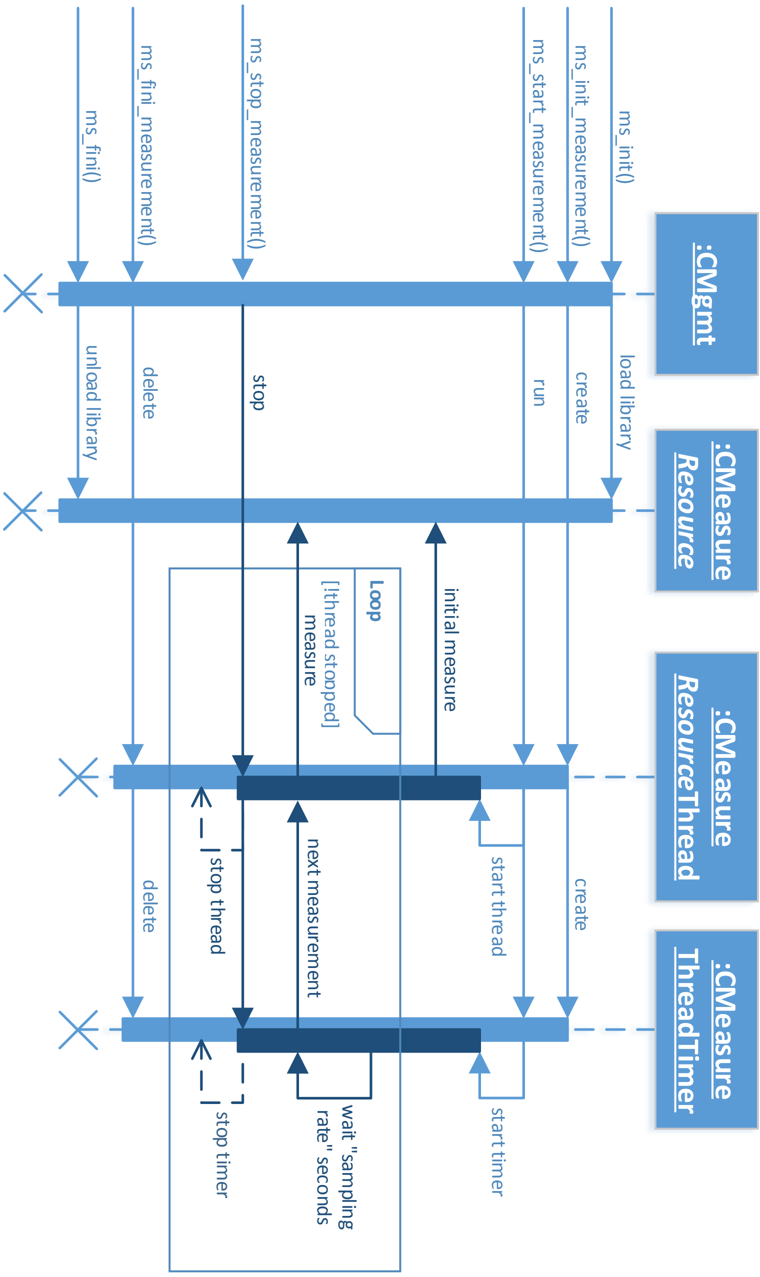

Figure 3: Libmeasure sequence diagram showing the interactions between different objects of the library. Please note that the figure illustrates a simplified representation.

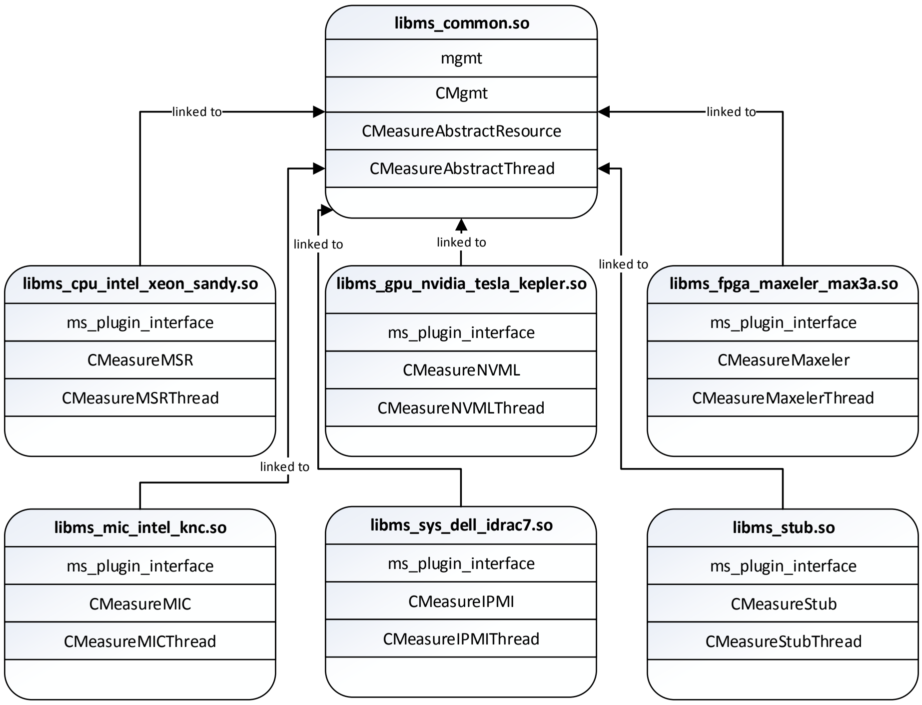

Figure 4: Overview of libmeasure with all available modules.

3.3 Modular Expendability

The existing modules fit perfectly to our heterogeneous node but it is very unlikely that someone uses a system with the same hardware resources. Therefore the modular software architecture can be extended by additional modules for new or different resources. In the following section, we explain how new modules must be implemented and what programmers have to modify in the `libms_common.so` to add a new module. Since you must modify the code of `libms_common.so`, this shared object is not really resource-independent at the moment. In one of our next future releases, we will address this issue, so that programmers must only provide a plugin without any changes to other code segments of Ampehre. The following list shows all files and classes of the libmeasure which have to be edited or added.

- **resource-specific module:**
  - `ms_plugin_interface`
  - `CMeasureResource`
  - `CMeasureResourceThread`
  - `CMakeLists.txt`
- `measurement.h`

- **libms_common.so:**
  - `interface.cpp`
  - `CMgmt`
  - `mgmt.cpp`

In the following sections, we discuss each modification in detail. Please feel free to use the existing modules as examples to create new modules.

3.3.1 Resource-specific module

For a complete module, we recommend to implement six source files and a `CMakeLists.txt`. Please, add a directory for each new module. In order to build your module together with our modules you should add the name of your module's directory to the `CMakeLists.txt` in the libmeasure folder by using the `add_subdirectory()` cmake instruction.

ms_plugin_interface is the implementation of the C interface to the module/-plugin. The interface is identical for every module and the corresponding unique header file is `/include/ms_plugin_interface.h`. The functions necessary to implement this interface are shown in Listing 9.

```
void* init_resource(void* pLogger, uint64_t* pParams);
void  fini_resource(void* pMeasureRes);

void* init_resource_thread(void* pLogger, void* pStartSem,
                           MEASUREMENT* pMeasurement,
                           void* pMeasureRes);
void  fini_resource_thread(void* pMeasureResThread);

void  trigger_resource_custom(void* pMeasureRes);

```

Listing 9: Interface that each plugin must implement.

The init functions create an object of the `CMeasureResource` or `CMeasureResourceThread` class and return it as a void pointer. The fini functions get these objects as void pointers and delete the objects. The additional parameter `uint64_t* pParams` of the `init_resource()` function can be used to pass an arbitrary number of parameters to the `CMeasureResource` object during the initilization. While the init and fini functions are mandatory for a new module, the function `trigger_resource_custom()` is optional and can be used to call any custom function for a specific resource. For example, we use this function to process a "force idle function" of the FPGA, which is used to reconfigure the FPGA with an empty bitstream. If you do not need the additional custom function add it anyway but keep the function body empty.

CMeasureResource extends the class `CMeasureAbstractResource` and therefore has to implement the methods shown in Listing 10

```
private:
    void init(void);
    void destroy(void);

public:
    void measure(MEASUREMENT *pMeasurement,
                 int32_t& rThreadNum);
```

Listing 10: Methods of the `CMeasureResource` class that programmers have to implement in order to support a new resource.

The method `init()` is used to initialize the libraries needed to obtain the measurement values from the resources. For example in our GPU module we initialize the NVML, open the device, store the pointer to the device for later queries and set the frequencies to the desired values. This allows it to query the measurement values after the initialization from the resource without any unnecessary overhead. Furthermore the `init()` method can be used to display the capabilities of the resource on which the measurements are performed. An example is our CPU module where all available CPU frequencies are displayed.
After the last measurement, all used libraries need to be closed and the allocated memory must be freed. Therefore we provide a `destroy()` method which is similar to `init()`. For example, in our GPU module we set the clock frequencies back to the default values and close the NVML environment.
The method `measure()` is called to obtain the current measurement values from the resource. Every value which is measured here needs to be stored in the `MEASUREMENT` struct which is passed as parameter to the method. The extension of the `MEASUREMENT` struct to store additional values is explained later. The second parameter of the `measure` method is the thread id of the thread which executes this method. The thread ids are used for a first debugging without professional debuggers such as gdb.

CMeasureResourceThread extends the class `CMeasureAbstractResource` and thus has to implement the `run()` method as shown in Listing 11.

```
private:
    void run(void);
```

Listing 11: Methods of the `CMeasureResourceThread` class that programmers have to implement in order to support a new resource.

The method `run()` is the centerpiece of the `CMeasureResourceThread` class. At the beginning, all values in the measurement struct related to the resource need to be set to zero, especially those which are used as accumulators. Moreover the method needs to follow the structure shown in Listing 12.

```
void CMeasureResourceThread::run(void) {
    mThreadStateRun     = true;
    mThreadStateStop    = false;
    mThreadNum = CThread::sNumOfThreads++;

    mMutexTimer.lock();

    // Set all values in the measurement struct to zero.
    ...

    mpMutexStart->unlock();
    mrStartSem.wait();

    // First initial measurement.
    mrMeasureResource.measure(mpMeasurement, mThreadNum);

    while (!mThreadStateStop) {
        /*
         * This mutex is used to synchronize the while
         * loop with a timer. The timer unlock the mutex
         * every sampling rate seconds.
         */
        mMutexTimer.lock();

        /*
         * Here the measure method of class
         * CMeasureResource is called.
         */
        mrMeasureResource.measure(mpMeasurement, mThreadNum);

        /*
         * Calculates the difference between the current time
         * and the last stored time and returns the result
         * in time_diff and time_diff_double. Note, that the
         * calculated time always differs from the sample
         * rate. We decided to have this second time
         * information to compute energy in a more precise
         * way.
         */
        calcTimeDiff(&(mpMeasurement->
                        internal.resource_time_cur),
                     &(mpMeasurement->
                        internal.resource_time_temp),
                     &(mpMeasurement->
                         internal.resource_time_diff),
                     &(mpMeasurement->
                         internal.resource_time_diff_double));

        /*
         * Calculate results here and store everything
         * in the measurement struct.
         */
        ...
    }

    /*
     * Calculate some additional results such as average
     * power dissipation on the base of consumed energy
     * and store the results in the measurement struct.
     */
    ...

    // Thread termination
    exit();
}
```

Listing 12: Scheme of the `run()` method that each `CMeasureResourceThread` must have.

The mutex `mMutexTimer` gets locked in every loop run. We have implemented a timer thread which is part of the `libms_common.so` to unlock this mutex dependent on the sampling rates. The timer thread is a member of the `CMeasureAbstractResource` class but should be configured as shown in Listing 13. Please replace the string of `setThreadName()` with a custom resource-dependent string. In addition, set the timer with a resource-dependent sampling rate stored in a variable in the `MEASUREMENT` struct. Finally, you must share the mutex of the `CMeasureResourceThread` class with the timer. This is mandatory as this mutex is used for synchronization as explained above.

```
mTimer.setThreadName("Resource timer");
mTimer.setTimer(&(pMeasurement->resource_time_wait));
mTimer.shareMutex(&mMutexTimer);
```

Listing 13: `CMeasureResourceThread` constructor template.

CMakeLists.txt compiles all source files to one dynamically loadable library and links the module against the provided `libms_common.so` library. Finally, as already mentioned above, you must add the directory containing the new module and the if statement for the new module to the `CMakeLists.txt` located in the `/libmeasure` directory. Code for the new module in the `CMakeLists.txt` should look like the already existing. We also need to add a new option for the module to the `CMakeLists.txt` of the project root directory as shown in Listing 3.

3.3.2 measurement.h

For a new resource a define preprocessor directive has to be added to the header `measurement.h`. We assign a single integer to each resource. The i-th resource gets the value 2^i, so that each bit of the integer indicates one specific resource. Moreover the new resource needs a define directive which indicates whether the library or the stub should be loaded. The define preprocessor directive in the ifdef-statement is triggered dependent on the options set in the `CMakeLists.txt` file of the project's root directory (Listing 3). This preprocessor directive `Resource_LIB` will only be true if the library is compiled with support for the corresponding resource. The edited header including the new resource could look like Listing 14.

```
#define CPU      0x01
#define ...
#define RESOURCE 0x20
#define ALL      (CPU | GPU | GPU | FPGA | SYSTEM | MIC | RESOURCE)

#ifdef Resource_LIB
    #define Resource_LIB_NAME "libms_resource.so"
#else
    #define Resource_LIB_NAME "libms_stub.so"
#endif
```

Listing 14: Extended `measurement.h` header file with new define directives to support a new resource.

The MEASUREMENT struct is also defined in measurement.h. For every value which should be measured, a new variable needs to be added here. Furthermore the methods to obtain the measured values after stopping the measuring procedure are declared here. For this, you have to add a function for each value of interest stored in the struct.
Temprorarily variables which are needed for the calculation of other measurement results can be added to the C struct MEASUREMENT_INTERNAL. Furthermore, you need to add new variables to the internal struct to store time stamps for the new resource. These timespec structs are used to hold precise time information and can be used to calculate energy out of measured power values. The additions to the internal struct should look like the code shown in Listing 15.

```
struct timespec resource_time_cur;
struct timespec resource_time_temp;
struct timespec resource_time_diff;
double resource_time_diff_double;
```

Listing 15: Extensions for the MEAUSREMENT_INTERNAL struct defined in measurement.h.

3.3.3 libms_common.so

You have to modify some files which are compiled and bundled to the libms_common.so library. The necessary modifications are described in the following paragraphs. libms_common.so should be resource-independent. For this, we will remove all resource-specific code from the library in one of the next releases of Ampehre.

interface.cpp implements the interface to obtain all measured values as well as data that are calculated in the CMeasureResourceThreads. For each of these values, a function declaration must be added to measurement.h and the corresponding function definition must be added to the interface.cpp file. These functions are used to return the measured values stored in the MEASUREMENT struct. The MEASUREMENT struct should be the first parameter of the functions. The following listing shows an example for a function returning the total/accumulated energy consumed during a measuring.

```
double gpu_energy_total(MEASUREMENT *measurement) {
    return measurement->nvml_energy_acc;
}
```

Listing 16: Example for a function to return values stored in a MEASUREMENT struct.

CMgmt is the link between the C user interface and the resource-specific threads and thus provides functions to control the measurement threads (take a look to Figure 3). The class CResourceLibraryHandler is used to dynamically load a module with dlopen(), initializes the resource-specific object CMeasureResource and stores it as a pointer. The CResourceLibraryHandler calls the plugin interface and provides functionality to execute any of the plugin interface functions via dlysm(). Each resource-specific module such as CMeasureResource is instantiated in CMgmt and is stored for later access. The CResourceLibraryHandler

gets the module name passed as parameter in the constructor. It automatically loads the module and instantiates a `CMeasureResource` object. The last parameter of the `CResourceLibraryHandler` constructor is used to pass any number of parameters as `uint64_t*` to the `CMeasureResource` class of the new module. For example, we use this parameter to set the GPU clock frequencies. The `CResourceLibraryHandler` object for the new modules is inserted to the data container `mResources` which is a member of the `CMgmt` class.
As shown in Listing 17, in order to add the new module a line of code has to be added to the constructor. In this example we pass the GPU frequency settings to the `CMeasureResource`. If no parameters are needed a `NULL` pointer should be used here.

```
uint64_t params[] = {gpuFrequency};
mResourceVector.insert(mResourceVector.begin() +
                       (int)log2(RESOURCE),
                       new NLibMeasure::CResourceLibraryHandler(
                       mLogger, RESOURCE_LIB_NAME, params));
```

Listing 17: Extension to the `CMgmt` constructor.

mgmt.cpp is the implementation of C interface to the library management functionality. Functions such as `ms_init_measurement()` and `ms_start_measurement()` (Section 3.1.3) which are declared in `measurement.h` are defined in the `mgmt.cpp` file. Three functions have to be modified in order to integrate a new resource to `libms_common.so` respectively libmeasure. The first function is `ms_set_timer` where the sampling rates for each resource are set. This defines how often the `measure()` function of the class `CMeasureResource` is called in the `run()` method of `CMeasureResourceThread`. Therefore the switch statement has to be extended by a new case as indicated in the next listing. The identifier of the case statement is defined in `measurement.h` (Listing 14 of Section 3.3.2).

```
case RESOURCE:
    measurement->resource_time_wait.tv_sec  = sec;
    measurement->resource_time_wait.tv_nsec = nsec;
    break;
```

Listing 18: Code to store resource-specific sampling rates in the `ms_set_timer()` function.

The second function which has to be modified is `ms_init_measurement()`. Here you must add an if statement which is needed for appropriate plugin instantiation. An example is listed in Listing 19.

```
if (flags & RESOURCE) {
    ms->initMeasureThread(RESOURCE, measurement);
}
```

Listing 19: Extension of the `ms_init_measurement()` function.

The third function which has to be modified is `ms_alloc_measurement()`. You should set the `struct timespec` which holds the sampling rate to maximum.

```
measurement->resource_time_wait.tv_sec     = UINT64_MAX;
measurement->resource_time_wait.tv_nsec    = UINT64_MAX;
```

Listing 20: Extension of the `ms_alloc_measurement()` function.

4 Hardware Requirements

The hardware requirements mentioned in this section are related to the resource-specific modules. For instance, if your GPU is manufactured by AMD, you cannot use our module, as the GPU module only works with Nvidia Tesla GPUs. Anyway, you are still able to use our measuring framework by disabling the GPU module (explained in Section 3.1.2).

4.1 System

- We obtain all system-related measurements via IPMI (Intelligent Platform Management Interface) utilizing a Linux kernel module accessible via the device file system entry `/dev/measure`.
- With IPMI we are able to get data from thermal and power sensors of both the motherboard (systemboard) and the power supply.
- Our server is a *Dell Poweredge T620*. Hence, in order to read some non-documented DELL features/sensors via IPMI, we implemented an additional wrapper library which composes raw IPMI messages for message exchanging with the BMC.
- Therefore, we guess that our measurement library can only work on **Dell** systems including **iDRAC 7** (and iDRAC 8?) BMCs.
- We compile the source code to a dynamically loadable module. The system module is named `libms_sys_dell_idrac7.so`.

4.2 CPU

- Most energy and thermal sensor data are stored in MSRs (Model Specific Registers) of the Intel RAPL (Running Average Power Limit) interface. We read the MSRs via our kernel module `/dev/measure`.
- The module is also used to collect some CPU utilization values.
- Additionally, we are able to set the CPU governor and other values such as minimum and maximum core frequencies via the GNU library `libcpufreq`.
- We deployed *two Intel Xeon E5-2609 v2* CPUs (microarchitecture: Ivy Bridge) to our server.
- As our library samples CPU registers which are model-specific, you can use our library only on systems with compatible CPUs. We guess that **Intel** CPUs with **Sandy Bridge**, **Ivy Bridge**, or **Haswell** microarchitectures should work well, if they **don't have an integrated graphics processing unit**. Integrated graphics are often available for consumer products such as the Core i3/i5/i7-processors.
- We compile the source code to a dynamically loadable module. The CPU module is named `libms_cpu_intel_xeon_sandy.so`.

4.3 GPU

- Measured values are retrieved by calling functions of the NVML (Nvidia Management Library).
- We deployed a *Nvidia Tesla K20c* to our system.
- All **Nvidia Tesla** GPUs with **Kepler** microarchitecture are supported (GK104, GK110, and GK210).
- We compile the source code to a dynamically loadable module. The GPU module is named `libms_gpu_nvidia_tesla_kepler.so`.

4.4 FPGA

- We utilize the MaxelerOS library to obtain power, temperature, and utilization.
- We deployed a *Maxeler Vectis* FPGA card to our system.
- Currently, our library only supports **Maxeler Vectis** (MAX3A) FPGA cards.
- We compile the source code to a dynamically loadable module. The FPGA module is named `libms_fpga_maxeler_max3a.so`.

4.5 MIC

- We use Intel's `libmicmgmt` MIC management library to obtain the measurements.
- We deployed a passively cooled *Intel Xeon Phi 31S1P*.
- All **Intel Xeon Phi** with **Knights Corner** (KNC) architecture should work well with the library.
- We compile the source code to a dynamically loadable module. The MIC module is named `libms_mic_intel_knc.so`.

Appendices

A Recommended Sampling Rates

Users must specify a sampling rate for each resource which is compiled as libmeasure module. The sampling rate defines how often measurement values are queried from the devices. Low sampling rates can produce substantial CPU load, since all the measurement threads are executed on the CPU. Hence, sampling rates have to be chosen carefully. Moreover, they have an impact on the accuracy of the measurement results and the CPU utilization. We have to find a trade-off between accuracy of the measurements and the CPU utilization which also leads to different CPU power consumption. Therefore we have methodically examined different sampling rate combinations using all five modules runnable on our heterogeneous system. We have measured the power consumption and utilization while all resources have been in idle state. The results are shown in Figure 5 and 6. Obviously, the CPU utilization and power consumption is highly dependent on specific system configurations. We hope that our results are helpful anyway.

Figure 5 shows the CPU utilization, sampling all sensors of all currently supported resources as specified in Section 4. The lines indicate our recommendations to achieve utilizations below a specific thresholds. For example, the blue line indicates that the utilization induced by our measuring library loading all resource-specifc modules stays below 2 %, if the sampling rates CPU: 40 ms, MIC: 50 ms, GPU: 40 ms, FPGA: 70 ms and System 100 ms or higher are used for the measurements.

Figure 6 shows the resulting CPU power consumption induced by sampling all sensors of all currently supported resources as specified in Section 4. Accordingly, the lines indicate what sampling rates have to be chosen to make sure that the CPU power consumption stays below specific thresholds. For example the sampling rates have to be CPU: 20 ms, MIC: 20 ms, GPU 30 ms, FPGA 40 ms, System 80 ms or higher to get a CPU power consumption of less than 18 W.

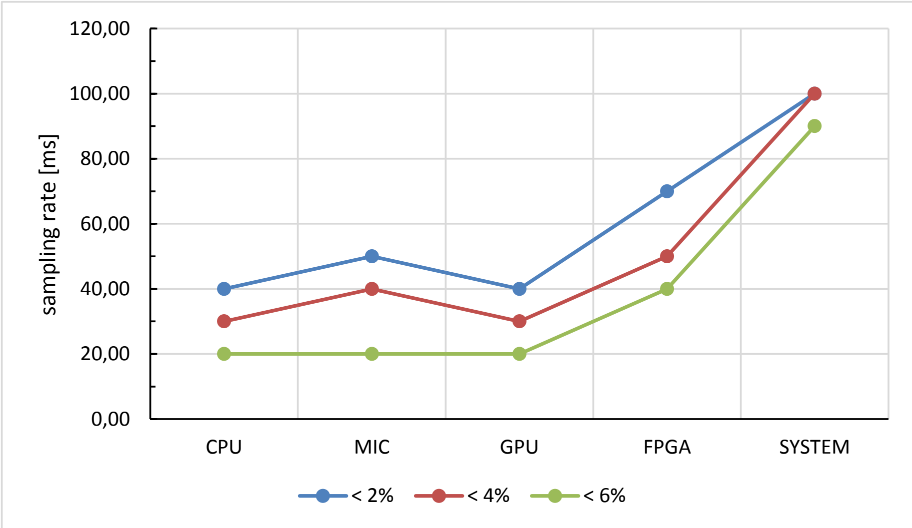

Figure 5: Libmeasure sampling rates and the resulting CPU utilization in percent. The lines indicate the lower boundaries of the sampling rates for which the CPU utilization is not higher than the corresponding threshold (2 %, 4 %, 6 %).

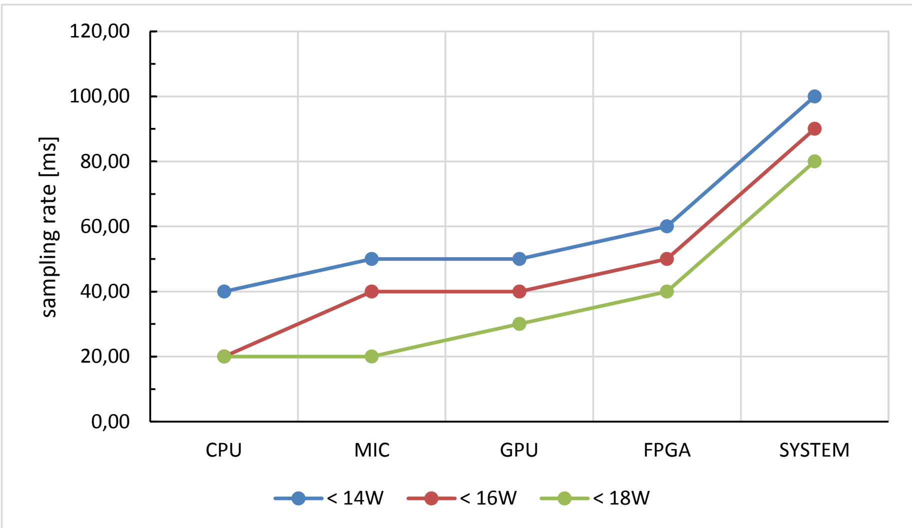

Figure 6: Libmeasure sampling rates and the resulting CPU Power consumption in Watt. The lines indicate the lower boundaries for which the CPU power consumption is not higher than the corresponding threshold (14 W, 16 W, 18 W).

B Utility usage

Each of our tools can be run with the “-h” option to show a help message where eyery command line option is briefly explained. The help dialog of hettime is shown in the following Listing.

```
$ hettime -h
Usage:
hettime [-h|-?|-i|-c SAMPLE_CPU|-g SAMPLE_GPU|-f SAMPLE_FPGA|-s SAMPLE_SYS|
         -G FREQUENCY|-C GOVERNOR|-L FREQUENCY|-H FREQUENCY|-o RESULT_FILE|
         -v CSV_FILE|-u] -e EXECUTABLE [-a "ARGS"]

-c SAMPLE_CPU   | Sampling rate for CPU power/temp measurements in
                | ms.
                | Default: 100ms. Recommended minimum: 20ms.
-g SAMPLE_GPU   | Sampling rate for GPU power/temp measurements in
                | ms.
                | Default: 100ms. Recommended minimum: 30ms.
-f SAMPLE_FPGA  | Sampling rate for FPGA power/temp measurements in
                | ms.
                | Default: 100ms. Recommended minimum: 50ms.
-m SAMPLE_MIC   | Sampling rate for MIC power/temp measurements in
                | ms.
                | Default: 100ms. Recommended minimum: 20ms.
-s SAMPLE_SYS   | Sampling rate for system-wide power/temp
                | measurements in ms.
                | Default: 100ms. Recommended minimum: 100ms.
-e EXECUTABLE   | Name of the executable. This option is mandatory.
-a "ARGS"       | Specify the arguments for executable EXECUTABLE
                | with this option.
                | Note that the arguments have to be seperated by
                | spaces. The arguments must be surrounded by
                | quotation marks! Note that the ARGS option
                | has to be the last in the argument list!
-G FREQUENCY    | Set a GPU frequency before the child application
                | get started.
                | Possible frequency settings are:
                | min, MIN, minimum, MINIMUM
                |    Set GPU frequency to its minimum value.
                | max, MAX, maximum, MAXIMUM
                |    Set GPU frequency to its maximum value.
                | cur, CUR, current, CURRENT
                |    Don't set GPU frequency.
                |    Leave the current setting untouched.
                | Default: cur.
-C GOVERNOR     | Set a CPU frequency scaling governor for the
                | 'acpi-cpufreq' driver.
                | Possible governors are:
                | save, SAVE, powersave, POWERSAVE
                |    Force CPU to use the lowest possible frequency.
                | dmnd, DMND, ondemand, ONDEMAND
                |    Dynamic frequency scaling. Aggresive strategy.
                | cons, CONS, conservative, CONSERVATIVE
                |    Dynamic frequency scaling.
                | Conservative strategy.
                | perf, PERF, performance, PERFORMANCE
                |    Force CPU to use the highest possible
                |    frequency.
                | Default: dmnd.
-L FREQUENCY    | Set the lowest permitted CPU frequency in MHz.
-H FREQUENCY    | Set the highest permitted CPU frequency in MHz.
-o RESULT_FILE  | Save results in a file instead of printing
                | to stdout.
-v CSV_FILE     | Save results in a CSV table file.
-u              | Use UNIX socket handler library to communicate
                | with msmonitor.
-i              | Forcing FPGA to idle after measuring system
                | initialization.
-h              | Print this help message.
-?              | Print this help message.

Example:
hettime -c 90 -i -G min -C conservative -e /usr/bin/find
        -a "/usr -iname lib*"
```