rtfparallella.docs Documentation

Mahmoud Bazzal

Nov 17, 2020

Contents:

CHAPTER 1

Introduction

Source coode for this project can be currently found in the eclipse APP4MC examples repository. Initial merge of this project with APP4MC examples repository can be found in this link.

Soruce code is also available on GitHub.

This file is intended to be an introduction that explains the main functionality of RTFParallella framework.

1.1 What is RTFParallella

RTFParallella is a framework that allows the easy implementation of nulti-core, real-time applications on the Adapteva Parallella hardware platform by using the 16-core Epiphany III co-processor on board and exploiting the Network on Chip (NoC) architecture of that chip for deterministic multi-core implementations of automotive software. The organization of implementation code within the framework as well as user code has been arranged according to Amalthea. models of a multi-core system (both hardware and software artifacts), The code will only require attributes of tasks and their deployment on cores as a tuple of Amalthea parameters.

1.2 Contribution

This project has been started in the context of Google summer of code. During Google Summer of Code the following has been contributed to RTFParallella:

- Fixed the FreeRTOS port for Epiphany III processor to be able to realize RMS schedulling.
- Relaized RMS scheduling policy for tasks on Epiphany processor.
- Adapted Amalthea model task structure to run using FreeRTOS.
- Added support for shared memory management and distributed shared memory management.
- Added visuallization of real-time behavior of amalthea model on EpiphanyIII processor.
- Added examples that could be changed (manually or by code generation) to realize any Amalthea model. (given it fits within the parallella hardware constraints).

1.3 Quick start

- Clone RTFParallella source code and examples from the APP4MC examples repository.
- Set up cross complilation environment as in *Cross compilation setup (steps):*.
- run *patch_init.sh*

```
sh patch_init.sh
```

- run *make all* from *src/parallella*
- set up the deployment script as in *Deployment Script*.
- deploy the binaries to Parallella (*.elf files*) and (*host_main*) using *parallellaDeploy*
- SSH into parallella board and run host_main

1.4 Documentation scope

This documentation will include the following:

- A step-by-step guide on the setup of the Adapteva Parallella Hardware platform.
- A guide on setting up the eclipse IDE for cross compilation nad deployment/ running of RTFParallella code on Adapteva Parallella.
- A guide showing the intended usage of the framework and how to manipulate such framework to realize the desired real time behavior.
- The design of RTFP and how it could be adapted to other hardware platforms.
- Limitations that should be taken into account when using RTFP.

1.5 Index

1.5.1 Parallella hardware

The Adapteva Parallella development board is designed to provide multicore functionality for high parallellism applications in a small form factor.

Adapteva Parallella achieves a high level of parallellism by using the the 16 core Epiphany III co-processor.

The Epiphany III co-processor is a network on chip (NoC) processor. Which performs message passing between different cores in deterministic time.

In order to access the Epiphany co-processor, the board includes an FPGA with a soft dual core, ARM v7 processor running on it.

Since the Epiphany co-processor is a bare-bones processor, no operating system is running on it. Therefore, many of the IO functionalities that are typical in any C program running on linux (or windows), such as pritf does not work. Because of this, IO of the Epiphany chip is done by simply writing the outputs to a DRAM shared memory buffer where it would be read by the ARM core running on the FPGA (since it runs linux).

Additionally, many of the Adapteva Parallella boards come without an HDMI connector, and therefore the only way to access it is by using SSH.

It is expectd that the user has already set up all prerequisites to establish SSH connection to the board before trying to use RTFP.

Notes

This board will overheat. A fan shoudl be used for cooling. If a fan is not avaiilable, place the board vertically to allow for air flow through the heat sink.

1.5.2 Adapteva Parallella setup

This chapter will detail the process of setting up all tooling required for development on the parallella board for RTFP.

This setup is intended to be run on a linux machine, The steps shown here have been realized on an Ubuntu 18.04 system, it should also work with any other distributions -and versions- of linux.

Dependencies:

RTFP requires the following dependencies to allow successful compilation and deployment of binaries generated using the framework:

- ESDK (Epiphany SDK) : v2016.11.
- Eclipse for C/C++ developers.
- Linaro cross GCC toolchain.

Cross compilation setup (steps):

This framework usually requires cross compilation on a host machnine (development machine) and deployment to the Parallella board.

This section will explain the setup for cross compilation. Deployment setup will be explained in a different section.

steps are as follows:

- Download The Epiphany SDK (ESDK) from Github. This framework has been written using the latest ESDK (v2016.11), it has not been tested on previous ESDK versions.
- The ESDK is released in 3 versions:
  - armv7l: for compilation on the parallella board (this version has an arm-compatible GCC).
  - x86-64: This is the version that was used for development of RTFP and it must be used for cross compilation of Parallella binaries on thelocal machine.
  - x86-64: This version could be used for simulating the operation of parallella on the local machine (requires at least 8GB of ram, for more details on that, refer to this repository).
- Create a directory at

```
/opt/adapteva
```

- unzip the tar.gz file here using the following command

```
tar xf esdk.2016.11.x86_64.tar.gz
```

- create a soft link for the ESDK folder, this step is optional, but wil simplify further operations as it is easier to type than the full name of the ESDK folder

```
sudo ln -s esdk.2016.11 esdk
```

- open the bash start up file

```
sudo nano ~/.bashrc
```

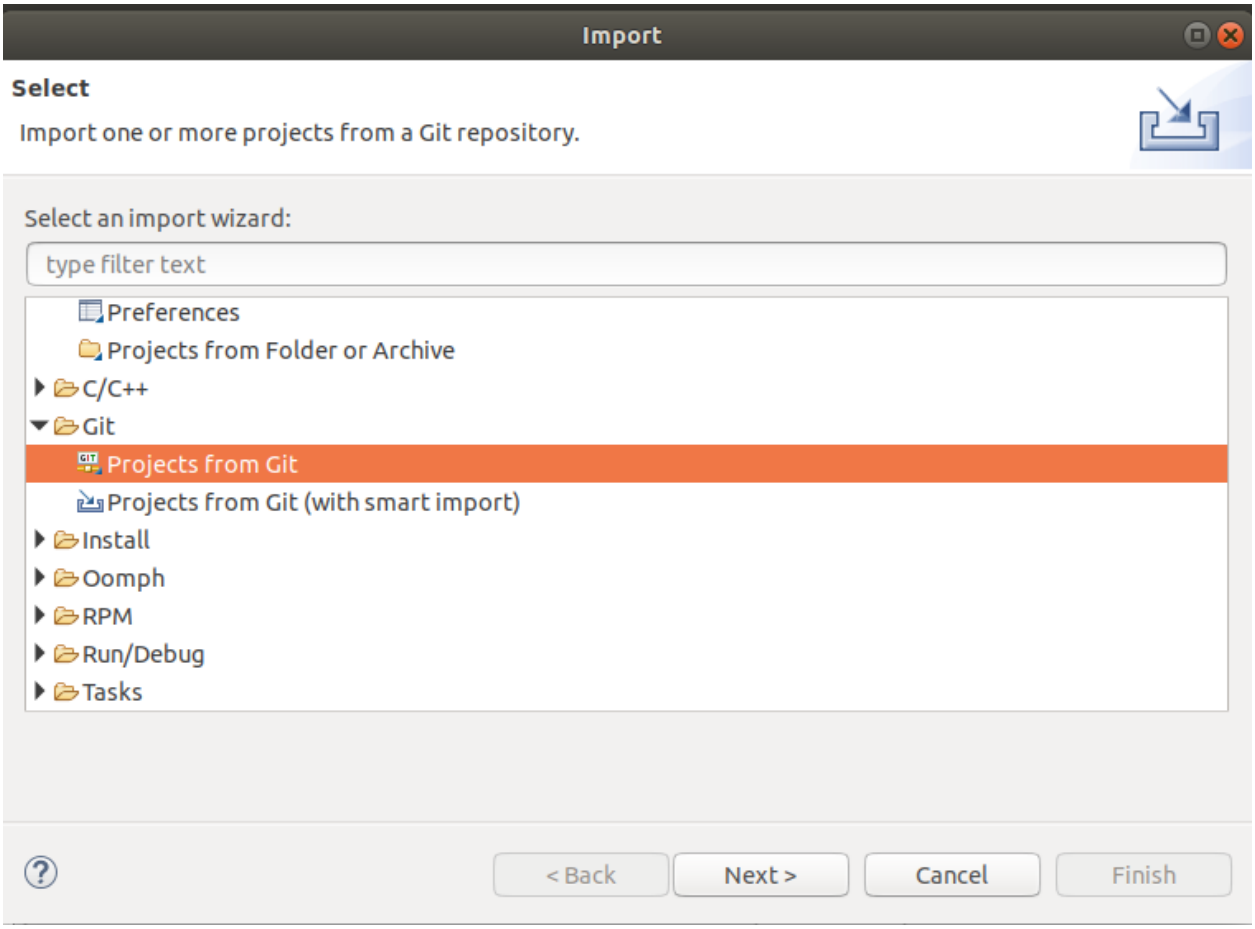
- Append the following to the end of the file

```
EPIPHANY_HOME=/opt/adapteva/esdk
. ${EPIPHANY_HOME}/setup.sh
```

- restart terminal (to allow bash to intialize again)
- For ARM code compilation, a gnubeahif cross compiler is needed. download the Linaro toolchain for cross compilation here.
- unzip the tool chain tar in the following folder

```
/opt/linaro
```

- Download Eclipse for C/C++ developers here.
- open eclipse
- File -> import -> git -> projects from git -> next



- enter the URL of RFTP repository and import the project.

- test the setup by trying to build the project.

Notes

It is important to run eclipse from the terminal (to make sure bash is initialized), to allow make file invcation from within eclipse to recognize e-gcc (GCC compiler for Epiphany processor).

1.5.3 Binary deployment on Parallella

This file explains the process of deplyment of software artifacts generated on a development machine to the Parallella board and running those artifacts on Parallella.

Deployment Script

The deplyment script can be found in the root folder of RTFParallella

```
/parallellaDeploy.sh
```

Deployment parameters

The deployment script has the following parameters. Those parameters are to be changed for the particular parallella board and development machine used :

- `HOST_NAME` : Replace with IP address or host name of parallella
- `HOST_USER` : Replace with name of the user with privilige to execute files in `HOST_OFFLOAD_PATH`.
- `HOST_OFFLOAD_PATH` : Replcae with desired execution path on parallella board file system (usually this will be a folder under `home`).
- `PORT` : Replace with the SSH port on parallella board (usually 22).
- `KEY` : replace with the absolute path to ssh the public SSH key file on the development host.

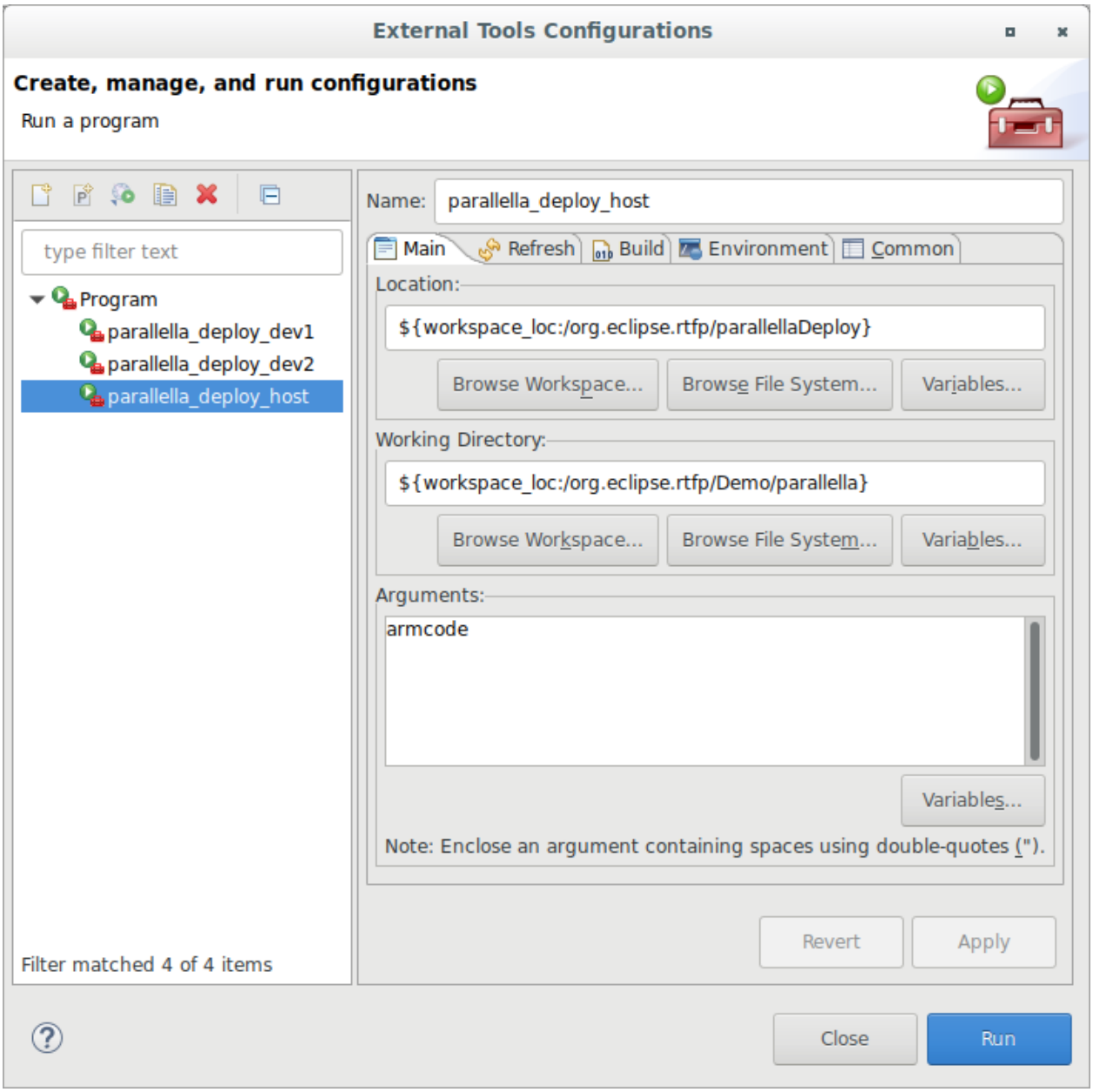
Invocation of Deployemnt script

This script must have at least one argument that must be passed when it is invoked. Namely, the file which should be deployed, any number of files is allowed, arguments (files' names) must be separated by a space.

This script will be called automatically by eclipse once the launch configuration is set up.

Eclipse launch configuration setup

The following steps show the process of setting up eclipse for deployment of binaries:



- Each deployed binary will have its own launch configuration
- Run -> External tools -> external tool configurations
- New Launch Configuration
- Choose the name of the deplyment configuration (usually parallella_deployment_device_coreNum or parallella_deployment_host)
- in Location, choose browse workspace
- choose the deployment script (in root folder of the project)
- in working directory, choose browse workspace
- choose the folder that contains the binaries (/Demo/parallella)
- in arguments, insert the name of the binary to be deployed with the extension (if any)

- save the configuration
- repeat the steps for all binaries.

Known issues

The development host's SSH key must be listed as a trusted key in the parallella board authorized key file

```
~/.ssh/authorized_keys
```

1.5.4 Real Time Framework Parallella

This page describes the core functionality of Real Time Framework Parallella, for example code and further technical explanation

Introduction

This product is a C framework to help in early prototyping of multi core systems based on AMALTHEA models. It will accept parameters from the model and realize it in C code which can be tested and benchmarked on existing hardware. As the name suggests, this framework is designed to run on the Adapteva Parallella platform. However, it is possible to adapt it to other platforms.

Platform dependencies

This framework has been bult using FreeRTOS. It uses task control blocks (TCBs), critical section APIs, and task handling APIs from FreeRTOS.

The FreeRTOS port for Adapteva Parallella has been first written here. That implementation has been modified to achieve tick accurate, reproducable task switching behaviour.

This framework uses Epiphany Software Development Kit (ESDK). the latest ESDK version (2016.11) was used.

How to use RTFP

RTFP is designed to be used with AMALTHEA models (constructed and analyze using APP4MC).

With this in mind, in order to use RTFP, the following steps can generally be used:

1. Amalthea model is constructed and verified in APP4MC using analysis plugins, visualization plugins... etc.
2. Tasks attributes are used to construct tasks in RTFP.
3. Task and memory operation code (copying labels, etc) are constructed.
4. Runnable code is constructed.
5. Shared memory is allocated statically.
6. RTOS tasks are created using the AmaltheaTask objects created in step 2
7. communication between the epiphany coprocessor and the host is defined (optional)
8. constarints are added on task status data to benchmark the system and verify real time aspects.

1.5.5 Task management in RTFP

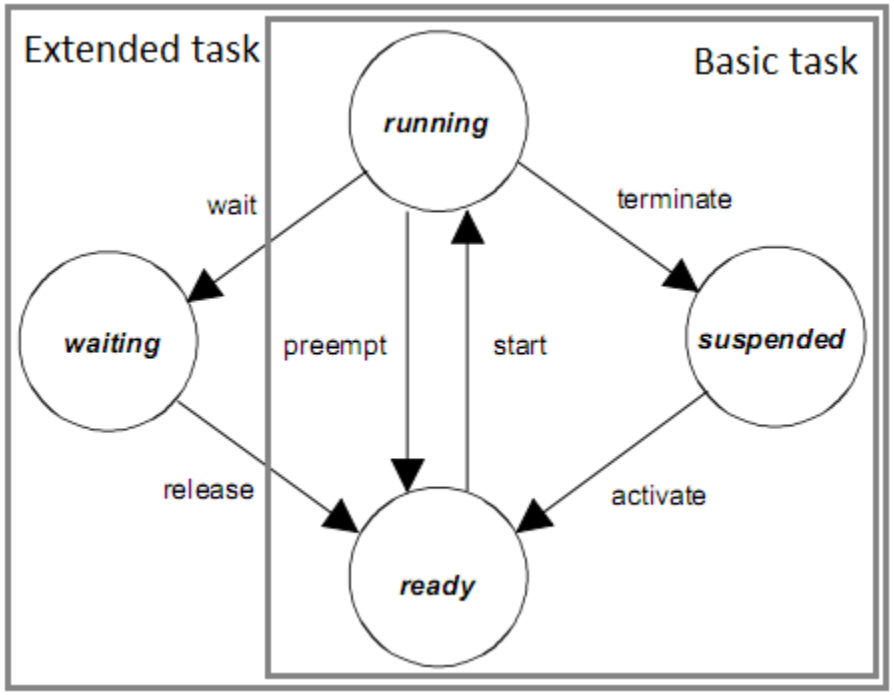
Overview

In real time multi-core systems, it is essential to realize separate tasks that perform -usually- independent actions. RTFP follows the OSEK standard to manage tasks.

Since it was based on FreeRTOS, RTFP has the semantics of OSEK within the FreeRTOS kernel, with different names

- Running
- Waiting
- suspended
- Ready

For more detailed description of task state model, refer to the OSEK specifications. This diagram shows the transition between task states.



RMS task scheduling

RTFP will schedule tasks according to RMS (Rate Monotonic Scheduling) policy by default.

Resterictions when using RTFP with RMS scheduling:

- priorities of all tasks are unique.
- the task with the smallest period must have highest priority.
- Priorities decrease by 1 each time for the other tasks depending on peiorities.
- Highest priority is number of all tasks running on a core.

Fixed priority task scheduling

As mentioned elsewhere, RTFP is based on FreeRTOS, and therefore, the original scheduling policy of FreeRTPOS could be used to manage tasks. FreeRTOS uses fixed priority schedling. In order to use this schedling, it is simply required to alter the priority scheme used for the tasks. While RMS requires all the tasks to have unique, monotonically increasing (or decreasing priorities), Fixed priority scheduling does not require such restrictions, multiple tasks could have the same priority, and different priorities are not assigned based on a task's period but is left for the descretion of the developer using RTFP.

Creating tasks using RTFP

To create a task in RTFP, first initialize an amaltheaTask struct as follows

```
AmaltheaTask createAmaltheaTask(void *taskHandler,
     void *cInHandler,
     void *cOutHandler,
     unsigned int period,
     unsigned int deadline,
     unsigned int WCET);
```

arguments:

- taskHandler : pointer to the function that contains task code
- cInHandler : pointer to the function that creates a copy of shared varialbles (labels) on the task stack. This function will be called at the beginning of every new instance of a task.
- cOutHandler : pointer to the functions that returns the local copy of shared variables into shared memory. (write operation).
- period : period of the task in ticks
- deadline : relative deadline in ticks
- WCET : Worst Case Execution Time in ticks.

This function will return a struct of type AmaltheaTask. This struct will be used to create an RTOS task as follows:

```
void createRTOSTask(AmaltheaTask* task,
        int priority,
        int argCount,
        ...);
```

Arguments:

- task : pointer to the AmaltheaTask struct
- priority : priority of the task (according to RMS, lowesrt perio has highest priority)
- argCount : number of different types of labels used by this task
- label_type_size : size (in bits) of label type.
- label_type_count: number of labels associated with that type.

1.5.6 BTF Specification

This chapter explain the basic structure and specification of Best Trace Format. It also provides the information about basic specifications incorporated in the tracing framework from the standard BTF Specification. The implementation of the BTF tracing framework is based on the specification defined in Eclipse wiki by Timing Architect.

BTF Structure

The BTF file consists of two parts:

1. All header section, containing information on objects of the trace and optional comments.
2. A data-section, containing the trace data of the simulation or measurement.

Header Section

The header includes parameters, used for the interpretation of the trace or information of the trace generator, and comments. Parameters and comments are indicated by a '#'-symbol.

The typical header of a BTF-file includes at least the version, creator, creation date and the time scale. Further information is optional.

```
#version 1.0
#creator
#creationdate 2019-06-18T22:25:05
#inputFile
#timescale ns
#entityType
#-0 T
#-2 R
#-9 SIG
#entityTable
#-0 [idle]
#-1 Task_ESSP3
#-2 Task_ESSP2
#-3 Task_ESSP5
#-4 Task_ESSP9
#-5 Task_ESSP8
#-6 Task_ESSP0
#-7 Task_ESSP6
#-8 Task_ESSP1
#-9 Task_ESSP4
#-512 Core_0
#-513 Core_1
#entityTypeTable
#-T [idle]
#-T Task_ESSP3
#-T Task_ESSP2
#-T Task_ESSP5
#-T Task_ESSP9
#-T Task_ESSP8
#-T Task_ESSP0
#-T Task_ESSP6
#-T Task_ESSP1
#-T Task_ESSP4
#-C Core_0
#-C Core_1
```

The above image demonstrates the BTF header section generated by the tracing framework implemented over RTFParallella. It shows the version, creator, creation date, time scale and Amalthea model file name used for for simulation over Adapteva Parallella. It also lists the entity type which corresponds to the events and lists the tasks and core information. This forms the metadata of the BTF header section.

Data Section

The trace information is represented in CSV format. Each line describes one event. The interpretation of one line depends on the event type.

The data section consists of line by line interpreted data. Each line has eight columns, whereas the last column is optional. A line contains the following elements:

<Time>,<Source>,<SourceInstance >,<TargetType>,<Target>,<TargetInstance>,<Event>,<Note>

```
0,Core_1,0,T,Task_ESSP0,0,start,0
0,Core_0,0,T,Task_ESSP3,0,start,0
1000000,Core_1,0,T,Task_ESSP0,0,terminate,0
1000000,Core_1,0,T,Task_ESSP4,0,start,0
1000000,Core_0,0,T,Task_ESSP3,0,terminate,0
1000000,Core_0,0,T,Task_ESSP5,0,start,0
2000000,Core_1,0,T,Task_ESSP4,0,terminate,0
2000000,Core_1,0,T,Task_ESSP6,0,start,0
2000000,Core_0,0,T,Task_ESSP5,0,terminate,0
2000000,Core_0,0,T,Task_ESSP9,0,start,0
3000000,Core_1,0,T,Task_ESSP6,0,terminate,0
3000000,Core_1,0,T,Task_ESSP1,0,start,0
3000000,Core_0,0,T,Task_ESSP9,0,terminate,0
3000000,Core_0,0,T,Task_ESSP2,0,start,0
4000000,Core_0,0,T,Task_ESSP2,0,terminate,0
4000000,Core_0,0,T,Task_ESSP8,0,start,0
4000000,Core_1,0,T,Task_ESSP1,0,terminate,0
5000000,Core_1,0,T,Task_ESSP0,1,start,0
5000000,Core_0,0,T,Task_ESSP8,0,preempt,0
5000000,Core_0,0,T,Task_ESSP3,1,start,0
6000000,Core_1,0,T,Task_ESSP0,1,terminate,0
6000000,Core_1,0,T,Task_ESSP4,1,start,0
6000000,Core_0,0,T,Task_ESSP3,1,terminate,0
6000000,Core_0,0,T,Task_ESSP5,1,start,0
7000000,Core_0,0,T,Task_ESSP5,1,terminate,0
7000000,Core_0,0,T,Task_ESSP9,1,start,0
7000000,Core_1,0,T,Task_ESSP4,1,terminate,0
10000000,Core_1,0,T,Task_ESSP0,2,start,0
8000000,Core_0,0,T,Task_ESSP9,1,terminate,0
8000000,Core_0,0,T,Task_ESSP8,0,terminate,0
10000000,Core_0,0,T,Task_ESSP3,2,start,0
11000000,Core_1,0,T,Task_ESSP0,2,terminate,0
11000000,Core_1,0,T,Task_ESSP4,2,start,0
11000000,Core_0,0,T,Task_ESSP3,2,terminate,0
```

The time corresponds to the nano second scale factor in order to comply with Eclipse Trace Compass.

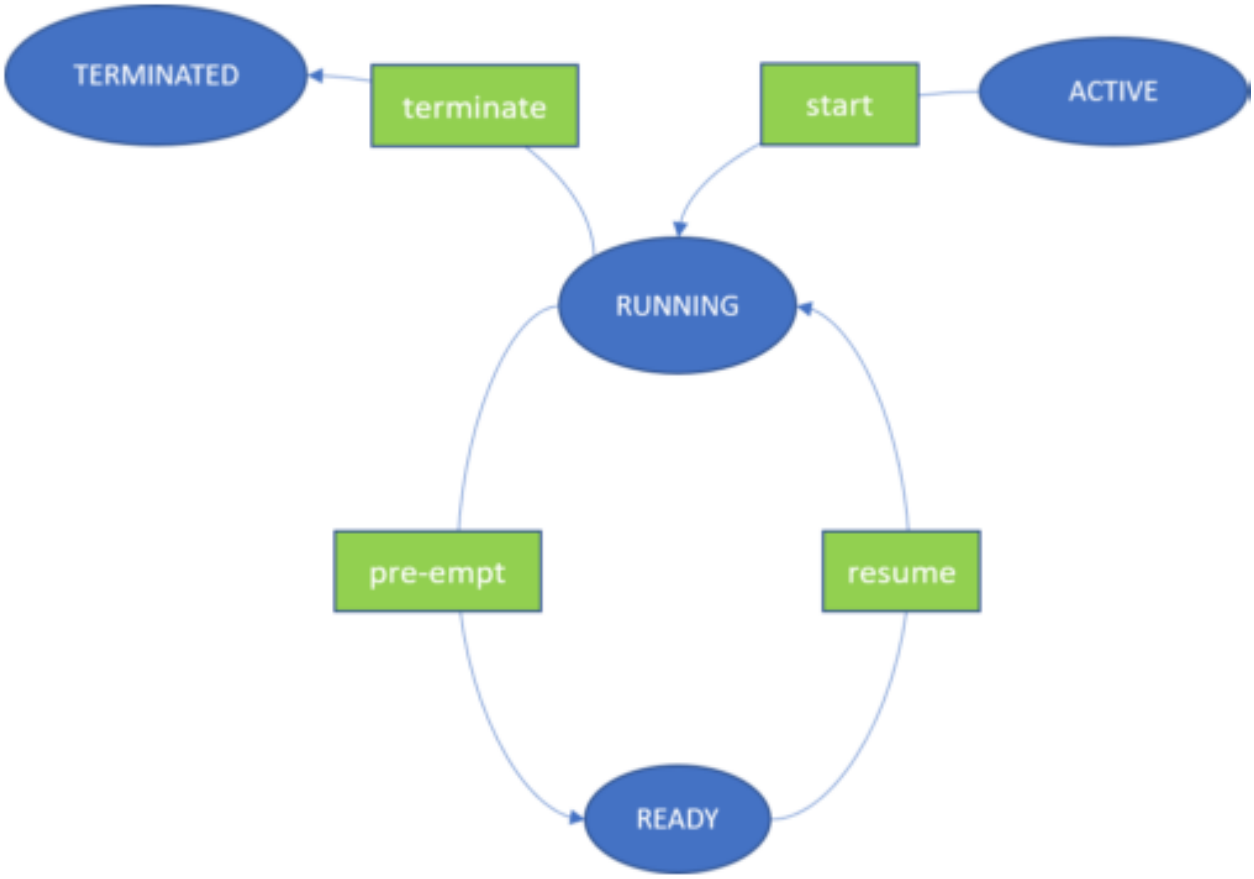
Supported Entities and Events

The BTF trace format supports multiple entities and events and is based on the OSEK Architecture. However, the developed tracing framework does not support the BTF specification in its entirety. This sections provides information about the supported entities and events in the tracing framework.

- PROCESS EVENTS (TASK and ISR EVENTS)

  A process can be either a task or an interrupt service routine. A scheduler assigns the process to a core where the process is executed. A running process can be preempted by another process which is of higher priority. The process resumes its execution once the higher priority task is executed.

Internal Event	Description	Source
START	Process instance is allocated to the core and starts its execution	Core
PREEMPT	Executing process instance is stopped by the scheduler	Core
RESUME	Preempted process instance continues execution on same or other core	Core
TERMINATE	Process instance has finished execution	Core



- RUNNABLE EVENTS

  A Runnable event is an atomic task executed within a Task Event. A task consists of multiple Runnables. A runnable is called within a process instance or in the context of another runnable.

Internal Event	Description	Source
START	Runnable instance is allocated to the core and starts its execution	Process
SUSPEND	Executing runnable instance is suspended as calling process preempts	Process
RESUME	Suspended runnable instance continue execution on same or other core	Process
TERMINATE	Runnable instance has finished execution	Process

- SIGNAL EVENTS

  A signal is a label, which can be accessed by a process instance.

Internal Event	Description	Source
READ	Signal is read by a process	Process
WRITE	Signal is written by a process	Process

1.5.7 BTF Tracing Framework

Overview

This section describes the BTF tracing framework over RTFParallella. The framework has been developed in C language. The framework however, is completely independent of the hardware platform and any system model simulation framework.

The tracing framework supports BTF trace generation of the tasks executed on a multicore heterogeneous platform. The current framework is restricted to task execution on two cores. However, this capablility can be enhanced based on the cores used to simulate the system model.

Compilation Steps

The framework has been been compiled using Epiphany Software Development Kit(ESDK) with the ESDK version being 2016.11.

The headers and the source file related to tracing framework can be compiled using any standard C compiler. The user can also create a static library and integrate it with their framework. The functionality of the tracing framework can be utilized by the APIs exposed in the framework header file.

Framework Implementation Details

This section describes the basic implementation details of the tracing framework. This enables the developer or the user to get the basic understanding of the framework in order to make a better utilisation of this framework.

As mentioned in *BTF Structure* the trace file consists of header section and data section.

The below enum defines the supported event type in tracing framework:

```
typedef enum btf_trace_event_type_t
{
    TASK_EVENT,
    INT_SERVICE_ROUTINE_EVENT,
    RUNNABLE_EVENT,
    INS_BLOCK_EVENT,
    STIMULUS_EVENT,
    ECU_EVENT,
    PROCESSOR_EVENT,
    CORE_EVENT,
    SCHEDULER_EVENT,
    SIGNAL_EVENT,
    SEMAPHORE_EVENT,
    SIMULATION_EVENT
} btf_trace_event_type;
```

The structure defined to hold and interpret the BTF data section is :

```
typedef struct btf_trace_data_t
{
    int32_t ticks;                  /**< Tick count */
    int32_t srcId;                  /**< Source Id */
    int32_t srcInstance;            /**< Instance of the source */
    int32_t eventTypeId;            /**< Type of event Runnable , Task etc.. */
    int32_t taskId;                 /**< Task Id */
    int32_t taskInstance;           /**< Instance of the task */
    int32_t eventState;             /**< State of the event */
    int32_t data;                   /**< Notes */
} btf_trace_data;
```

The supported events of the BTF specification are held in enum as defined below:

```
typedef enum btf_trace_event_name_t
{
    INIT = -1,
    PROCESS_START,
    PROCESS_TERMINATE,
    PROCESS_PREEMPT,
    PROCESS_SUSPEND,
```

(continues on next page)

(continued from previous page)

```
    PROCESS_RESUME,
    SIGNAL_READ,
    SIGNAL_WRITE
} btf_trace_event_name;
```

Following APIs are used to generate the header section of the trace file:

```
void write_btf_trace_header_config(FILE *stream);
```

The above function is reponsible for writing the mandatory header section to the BTF trace file. It includes the version, date of creation, input model file name and time scale.

```
void write_btf_trace_header_entity_type(FILE *stream, btf_trace_event_type type);
```

The above function writes the supported entity type in the BTF trace file.

To add the entry for entity table the following function can be used.

```
void write_btf_trace_header_entity_table(FILE *stream);
```

The entity type table can be dumped into the trace file using the below function.

```
void write_btf_trace_header_entity_type_table(FILE *stream);
```

The detailed documentation of the BTF trace framework APIs can be found here.

1.5.8 BTF traces in RTFParallella

This section explains about the fixes and enhancements added to the existing RTFParallella framework. Several hooks have been added to the applications running on the Epiphany core and the Host core processor.

Overview

The capability of the RTFParallella framework has been enhanced by adding the BTF tracing framework. The current RTFParallella framework operates on two core and executes 3 tasks and 2 tasks respectively on each core. The task execution follows RMS scheduling. Addition of tracing functionality adds some overhead to the overall simulation of the system model. This overhead cannot be avoided as it takes some extra clock cycles to dump the trace information in the allocated memory area. The initial Worst Case Execution Time(WCET) was defined based on the initial requirements which did not have any scope of adding any tracing framework. Therefore, the WCET has been slightly modified to simulate the Amalthea task model along with the BTF tracing functionality.

Feature Enhancements

- The initial implementation of RTParallella registered the timing information based on the tick count. This has now been replaced with the actual timing information. The default time scale is *ns*.
- The time taken per tick count on the Epiphany cores is now made configurable. The user can pass the time scale factor to configure the tick count. Currently, the tick count has been restricted to the 100 us and 1000 us.
- The visualization of the shared label values in the legacy RTFParallella trace was not showing the correct value, this has now been fixed.
- Synchronization between the epiphany cores as well as between epiphany core and host core processor is achieved using mutex implementation and software interrupts.

- The RTFParallella Config Header file can be used to configuring the memory sections. However, defining the memory sections must conform to the linker script used in the application.
- Addition of the BTF tracing framework.

Adapdation on RTFParallella

This section covers the information about the utility functions that has been used to adapt the tracing framework to the RTFParallella. Similiar approach can be used to integrate the tracing framework on other platforms.

The below structure hold the information about the BTF trace to ensure proper synchronization within host and epiphany cores.

```
typedef struct btf_trace_info_t
{
    int length;                                 /**< To ensure that the mutex is␣
↪initialized */
    unsigned int offset;                         /**< Mutex declaration. Unused on host  */
    unsigned int core_id;                        /**< BTF trace data buffer size which is␣
↪to be read */
    unsigned int core_write;                     /**< Read write operation between␣
↪epiphany core and host */
} btf_trace_info;
```

The supported entities in the BTF trace framework are Tasks, Runnables, Shared labels or Signals and Hardware Cores. The IDs of each entity must be defined in the RTFParallella config header file. The current ID values are allocated as:

- 0-15 integer value is reserved for the task ID.
- 16-63 integer value is reserved for the Runnables ID.
- 64-255 integer value is reserved for the Shared labels.
- 256 onwards is reserved for the hardware cores.

However, it is upto the developer to define the ID sections. Here is an example for this:

```
typedef enum entity_id_t
{
    /* 0 to 15 entity ID is reserved for TASKS. */
    IDLE_TASK_ID = 0,
    TASK5MS0_ID,
    TASK10MS0_ID,
    TASK20MS0_ID,
    /* 256 to 264 reserved for HARDWARE */
    HW_CORE0_ID = 256,
    HW_CORE1_ID
} entity_id;
```

Adapting Host Application

The host application is a normal linux executable file running on the ARM core. The command line arguments passed to the host executable is used to generate the BTF trace file along with the neccessary header information. The trace file is generated in the present working directory with the file name provided by the user. After creating the file, the BTF header is contructed using the below function:

```
static void construct_btf_trace_header(FILE *stream);
```

The above function write the mandatory header section, the supported entity type, entity table and entity type table. Each entity table must be generated based on the Amalthea task model. The model specific source file must holds the information about the Amalthea task model. This an example of the task model used in the RTFParallella:

```
static const char task_enum [][LABEL_STRLEN] =
{
    "[idle]",
    "Task5ms0",
    "Task10ms0",
    "Task20ms0",
    "Task10ms1",
    "Task20ms1"
};
```

The sequence of the tasks, runnables, hardware cores, shared labels must match to the sequence of the IDs created as an enumeration type in teh RTFParallella config header file.

The function parsing the command line arguments returns the scale factor which defines the time taken by each tick count on the epiphany core. The Epiphany core operates at a frequency of 700 MHz, the time taken by each tick count decided based on the scale factor and the operating frequency. The scale factor is stored in the shared memory which can be read by the Epiphany cores to adjust the timing of their tick count. The epiphany applications are then loaded on the respective cores and the host processor reads the shared memory area allocated for the trace metadata in order to read the actual BTR trace information. If any BTF data is read from the shared memory, this raw data is dumped in a temporary text file. Once the application processing is complete, the content of this text file is used to interpret and generate the BTF trace file which can be viewed by tools such as Eclipse Trace Compass.

Adapting Epiphany Application

Each Epiphanny core runs FreeRTOS, which is capable of scheduling the tasks. RTFParallella has the capablility to schecdule in task based on the Rate Monotonic Scheduling(RMS) concept. In order to generate the BTF trace, the structure of the Amalthea task model has been modified to include the source instance and source ID.

```
typedef struct AmaltheaTask_t
{
    unsigned int src_id;
    unsigned int src_instance;
    unsigned int task_id;
    unsigned int task_instance;
    void(* taskHandler)(int  src_id, int src_instance);
    unsigned int executionTime;
    unsigned int deadline;
    unsigned int period;
    void(* cInHandler)();
    void(* cOutHandler)();
}AmaltheaTask;
```

The function which generates the Amalthea task model has also been modified as below:

```
AmaltheaTask createAmaltheaTask(void *taskHandler, void *cInHandler, void␣
↪*cOutHandler,
    unsigned int period, unsigned int deadline, unsigned int WCET,
    unsigned int src_id, unsigned int src_instance, unsigned int task_id, unsigned␣
↪int task_instance);
```

Apart from the previous arguments, this function also takes the argument for source ID of the task, source instance of the source, task instance and task ID.

The function that writes the BTF trace to the shared memory area is defined below:

```
void traceTaskEvent(int srcID, int srcInstance, btf_trace_event_type type,
    int taskId, int taskInstance, btf_trace_event_name event_name, int data);
```

The synchronization between the epiphany cores as well as between the epiphany core and host core is fulfilled using the below function:

```
void signalHost(void);
```

This function make use of Epiphany SDK mutex implementation and software interrupt to achieve the synchronization.

The function details and the complete documentation of code can be found at this link.

1.5.9 Shared memory management in RTFP

Overview

Adapteva Parallella has a 1GB DRAM that includes a shared section which acts as a shared memory. This section is accessible by the Epiphany processor and the host arm processor as well. This DRAM section will be used to store variables (labels) that are shared between tasks.

Shared labels between tasks can also be stored on the local memories of epiphany cores. This mode of storage is discussed in the next chapter of this documentation.

shared variables are allocated in the shared memory statically.

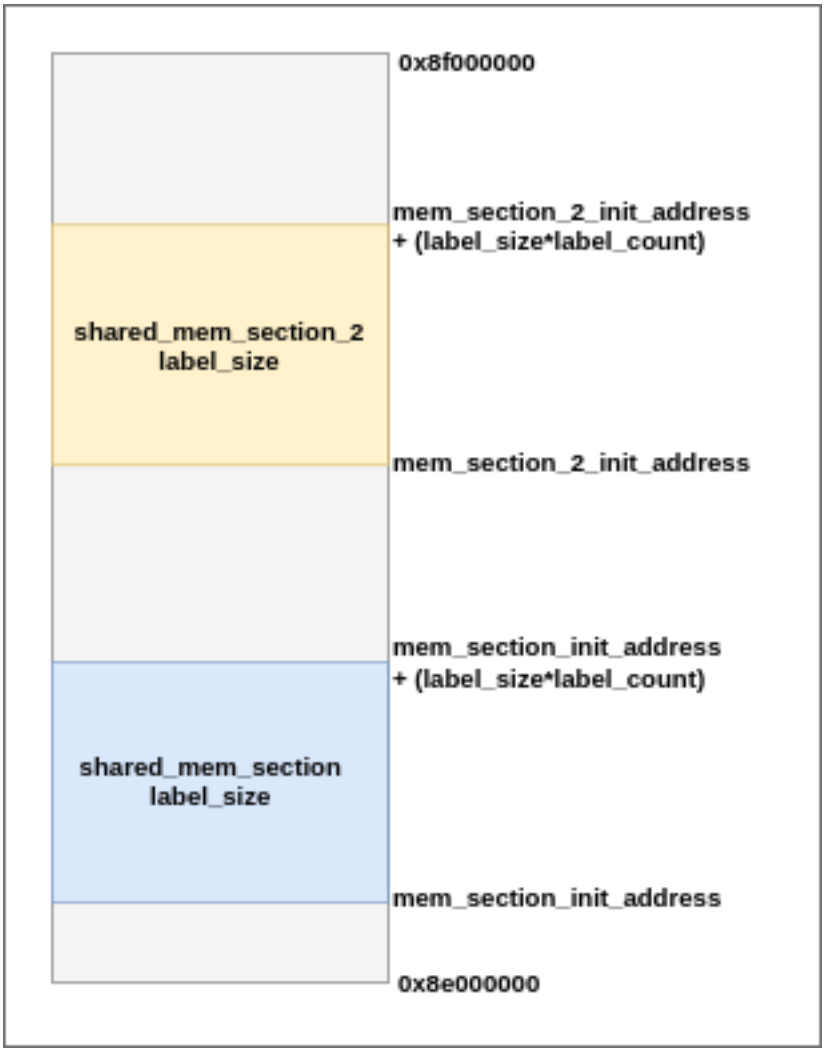
The labels are grouped together by size in contiguous memory blocks, and can be used as elemnts in an array.

Shared memory model

Shared memory address space spans 16 MB of the DRAM. within this address space, all shared labels should assigned. Shared memory between the epiphany processor and ARM host starts with address `0x8e000000` and ends with address `0x8f000000`.

All labels of the same type will be allocated to an array of that type. Accessing those labels can be done by simply accessing the index of the array.

This figure shows the memory model of DRAM on parallella.



Shared memory initialization and allocation in RTFP

Each shared memory section will be initialized individually. To intialize a new shared memory section in RTFP, the following steps are required:

- Declare a struct of the type `SHM_section`, Example:

```
SHM_section example_sec = {0x01000000,10,INT_32};
```

In this example, a block of 10 labels, each of which is of size unsigned int has been declared, the base address of this section is `0x8f000000`. Note that addresses given here are relative to the jointly accessible RAM section on the parallella and offsetted by `0x01000000`. i.e. base adresses could range between `0x01000000` and `0x01ffffff`.

Similarly, any other type could be declared. For label blocks that are too large to be declared as a standard C type, blocks of structs can also be declared.

- Allocate the declared memory block (array) in shared memory using `shm_section_init`, example:

```
//Declare a struct with section attributes
SHM_section example_sec = {0x01000000,10,INT_32};
//allocate the section in memory
```

(continues on next page)

(continued from previous page)

```
//and assign a pointer to it
unsigned int* sec_global_pointer = shm_section_init(example_sec);
```

After performing this operation, the pointer `sec_global_pointer` points to the base address of this section and can be used to access any value within the section by index. (similar to array accesses).

Shared memory write operation in RTFP

Declared memory sections in RTFP are accessed by their pointers. In order to write to a given label in a section:

```
//write to shared label
void write_shm_section (unsigned int* x, unsigned indx, int payload);
```

Where:

- `x` is the pointer to the declared section.
- `indx` is the index of the label being written to. Indices start from zero.
- `payload` is the value to be written.

Shared memory read operation in RTFP

A read operation is similar to the write operationn described above. Only the section pointer and indes are needed for the access.

```
int read_shm_section (unsigned int* x, unsigned indx);
```

Where:

- `x` is the pointer to the declared section.
- `indx` is the index of the label being read. Indices start from zero.

This function returns the value of the shared label as an integer. The return type is used for simplicity

known issues

- Due to the semantics of task to task communication in Amalthea models, a copy of every shared label will have to be created at the beginning of the task. However, the stack size of every task is limited and therefore on certain Amalthea models, it might be required to adjust the task's stack to prevent stack overflow.
- Access operations to the shared memory are not (yet) synchronised in RTFP. Race conditions may happen. This will be resolved in the next update of RTFP. with support for binary semaphores across cores on the Epiphany chip.

Future developments

In the next release of RTFP, the following functionalities will be added to shared memory management:

- Allocation of memory section will be done with the use of function calls instead of creating a pointer array. Each section will have a string identifier to refer to it throughout the code.
- Support for synchronisation between memory accesses on single core and accross multiple cores will be added.

- Automatic allocation mechanism will be added to insure that sections are contiguous and hence avoid memory fragmentation.

1.5.10 Distributed shared memory management in RTFP

Overview

Each core on the Epiphany processor has independent local memory which is accessible by other cores on the epiphany chip as well as the host AR< processor, and hence it can be regarded as distributed shared memory. This ditributed shared memory will be used to store variables (labels) that are shared between tasks. Additionally, it will be accessed regularly by the host ARM processor to get the status of the core and information on its operation at any given point in time.

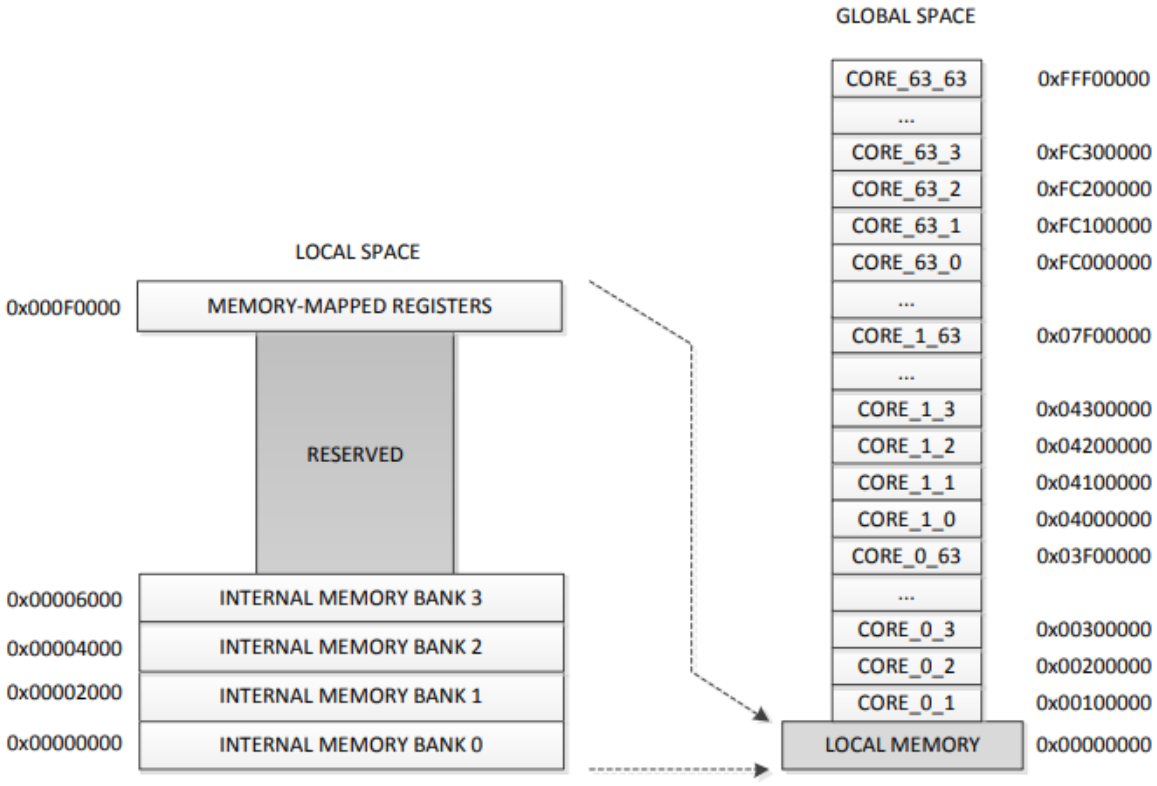
shared variables (labels) are allocated in the distributed shared memory statically.

The labels are grouped together by size in contiguous memory blocks called “sections”.

Distributed shared memory model

User accessible memory address space on core ranges from core_address + 0x0000 to address core_address + 0x7FFF with core_address referring to the start address of the core in the global address space, within this address space, all shred labels should assigned. This memory is divided into 4 memeory banks. Access to those memory banks can be restricted individully.

This figure shows the memory model of Epihany cores, the memory map is explained in more detail here.



All labels of the same type will be allocated to an array of that type. Accessing those labels can be done by simply accessing the index of the array.

Distributed shared memory initialization and allocation in RTFP

Each distributed shared memory section will be initialized individually. If the code using RTFP is automatically generated, initialization function should be called for each section individually.

To intialize a new shared memory section in RTFP, the following steps are required:

- Declare a struct of the type `DSHM_section` that contains the attributes of the memory section such as base address (relative to the core address space), nummber of labels, data type of the labels, and the row and column of the core where this memory is allocated. This struct is defined in RTFP as follows:

```
struct{
        unsigned                row;
        unsigned                col;
        unsigned int    base_addr;
        unsigned                label_count;
        TYPE                    sec_type;
}typedef DSHM_section;
```

For instant, a section of 10 labels is declared on core (0,0) with the base address of this section at `0x4000` as follows:

```
DSHM_section sec1_core_00 = {0,0,0x4000,10};
```

Note that even though the section type `sec_type` is a field of the struct `DSHM_section`, it can be omitted from the struct declaration, which indicates that the data type of this section is the default data type `unsigned int`.

- Allocate the declared memory section in the distributed shared memory by calling the function `DSHM_section_init`, example:

```
DSHM_section_init(sec1_core_00);
```

The return type of this function is void. It does not return a pointer to the memory section as is the case with shared memory. The reason for this is to allow for a more compact code that does not require calling this function to initialize a section on a core from multiple cores (including that core itself) in order to acquire the pointer to this section. This issue will be fixed by the next release of RTFP parallella such that a core can notify other cores on the epiphany chip of the existence of the section being initialized. This will simplify the process of distributed shared memory access accross cores. The current procedures for accessing such memory is described below.

shared memory access in RTFP

Declared memory sections in RTFP are pointers to actual memory addresses. In order to write to a given label in a section, the relative address in the core will be given by the index and that will be added to the base address of the core memory to find the absolute address of the variable (label) to be written.

```
unsigned int *addr;
unsigned int* addr_base;
addr_base = get_base_address_core(row,col);
addr = (unsigned int*) ((unsigned ) addr_base | (unsigned)outbuf_dstr_shared[label_
↪indx]);
*addr = payload;
```

Similarly to read the label:

```
unsigned int *addr;
unsigned int* addr_base;
addr_base = get_base_address_core(row,col);
```

(continues on next page)

(continued from previous page)

```
addr = (unsigned int*) ((unsigned ) addr_base | (unsigned)outbuf_dstr_shared[label_
↪indx]);
return *addr;
```

Where `*addr` will return the value at the requested label_indx.

In order to access the declared memory section anywhere in the project, the functions `write_DSHM_section` and `read_DSHM_section` can be used, in a similar way to accessing the sharede memory.

```
uint8_t write_DSHM_section(DSHM_section sec,int label_indx,int payload);

unsigned int read_DSHM_section (DSHM_section sec, int label_indx);
```

where `sec` is the struct that contains the section attributes, `label_indx` is the index of the label relative to the beginning of the section, and `payload` is the value to be written to memory (if any).

The read function currently returns the value stored in the memory address as an `unsigned int` by default. This will be changed in the next release of RTFP to allow for returning a type that is conformant with the memory section data type.

1.5.11 Memory Mapping

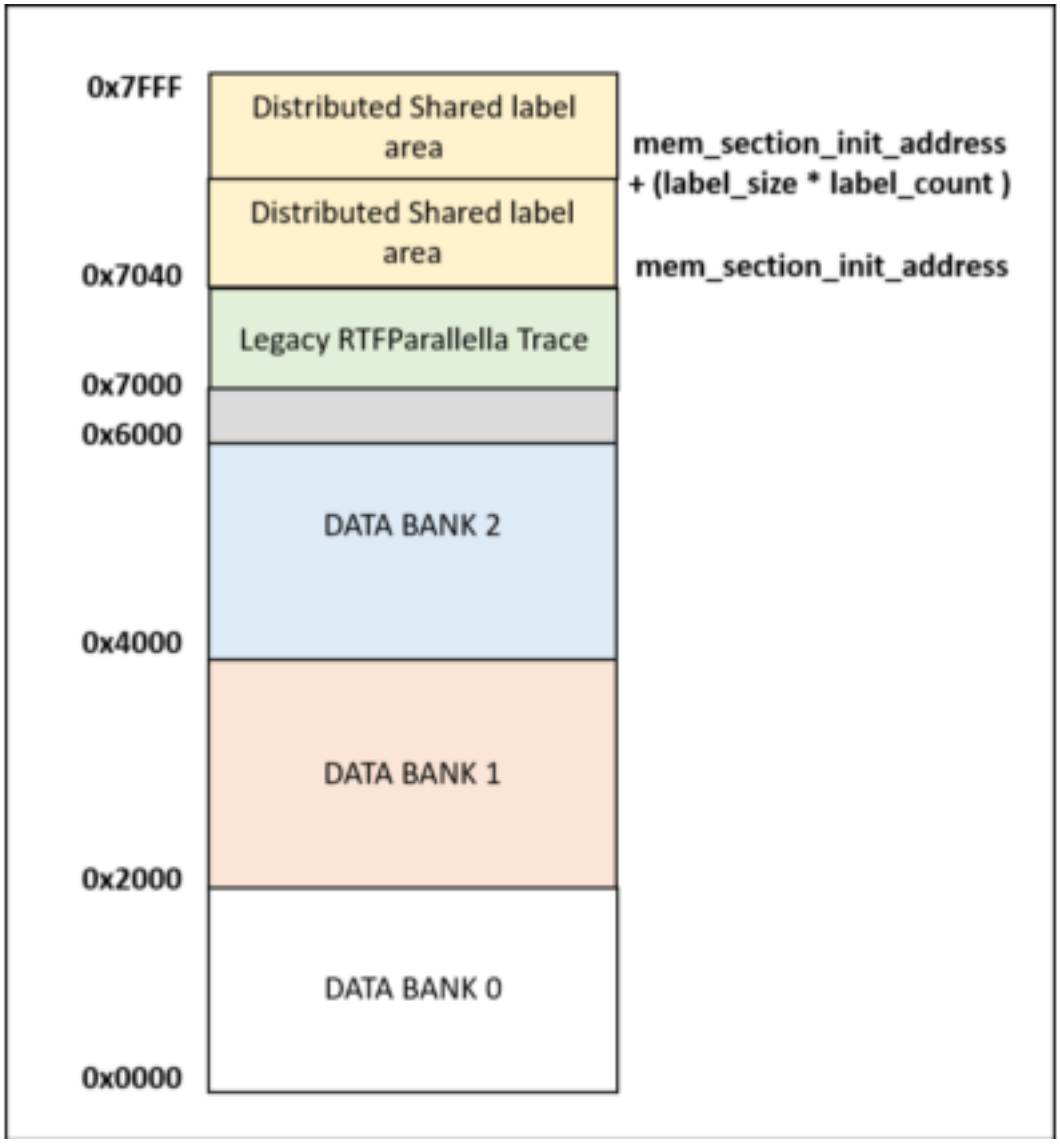
The Epiphany architecture uses a single, flat address space consisting of 2^32 8-bit bytes. However, the overall memory architecture on Parallella is divided in two sections. One which which is internal to the each core and the other which is shared among the cores.

Internal Memory

Each Epiphany core has an internal memory of 32 KB. This memory section is further divided into four memory banks. The memory section starts with the offset of 0x0000 and extends until 0x7FFF. Each memory bank is equally divided and comprises of 8 KB. The memory access between core to core is also possible by using the global address space. The first 3 bytes in the address space defines the core id of the epiphany core followed by 5 bytes of local address. Therefore, the global address space of each core range from 0x???00000 - 0x???07FFF. The same address range can be used by the Host processor to access the Epiphany core memory.

The trace framework has defined the address space for creating an application on top of it. It is recommended to use this address space when any application is developed on RTFParallella with the tracing framework. However, the memory mapping can be configured by changing the memory offsets in the RTFParallella config header file.

The below image shows the address mapping for the RTFParallella with tracing framework enabled:



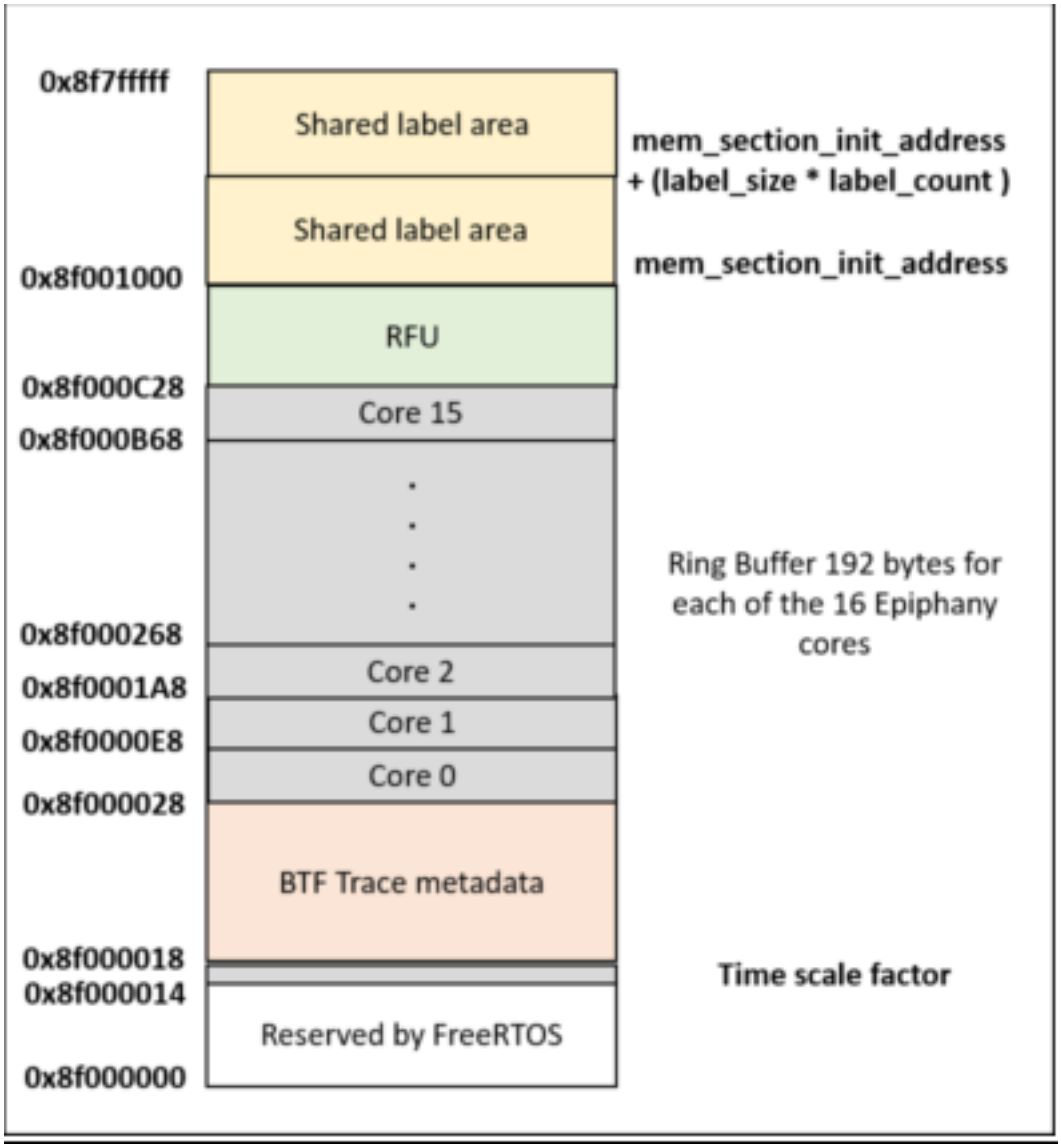
Besides, the below API can be used to get the memory pointer from the address space of Epiphany core in the application code of the Epiphany core:

```
unsigned int *allocate_epiphany_memory(unsigned int offset);
```

External Memory

The external memory is defined in the address range of 0x8E000000 - 0x8FFFFFFF. The size of the external memory is 32 MB. The first 16MB of memory region is reserved for the C libraries, code and data stack. The next 16 MB is further divided into two memory sections of 8 MB each. They form the *shared_dram* and *heap_dram* section.

In the implemented framework, the memory region of *shared_dram* is used to for storing the BTF trace and shared label data. The below figure demonstrates the memory region of shared_dram area:



The below API can be used to get the memory pointer from the address space of *shared_dram* memory region in the application code of the Epiphany core:

```
unsigned int *allocate_shared_memory(unsigned int offset);
```

Notes

The information provided above also depends on the linker script that is being used. The defined memory address space is based on the *fast.ldf* linker script.

1.5.12 Utility functions - task tracing

Overview

Since RTFP runs a RTOS on the epiphany processor, it is not possible to output debugging and tracing messages in real time.

The solution to thiis problem is writing specific debug flages to a memory buffer (which can be done in real time), and reading that memory buffer continuously from by the ARM host which can interpret and dispay these flages.

To summarize, ARM host will be polling the debug buffer for timely information on the current state of every Epiphany core separately.

Tracing running taks

to trace a running task, the following function can be called at any point during a task's execution.

```
void traceRunningTask(unsigned taskNum);
```

To trace the instance of a task, the following function can be called at the release of a task.

```
void traceTaskPasses(unsigned taskNum, int currentPasses);
```

Where currentPasses is the current instance of the task identified by its number taskNum.

Tracing time

To trace time progress in the Epiphany processo rwhen running RTFP, the following function has been added to the tick interrupt such that the tick changes will be traceable as soon as the tick interrupt handler is called. This will ensure tick accuracy of the time readout.

```
void updateTick(void);
```

To capture time progress on the ARM host side, the loop polling debug buffer should be timed to the same tick period set up in RTFP. The function

```
int nsleep(long miliseconds);
```

can be used to time the loop to make sure the epiphany is traced fast enough while no undersampling occurs.

Debug traces

The following function will write a user specified debug code to the debug buffer

```
void updateDebugFlag(int debugMessage);
```

It can be used to troubleshoot the system in case of failure to switch tasks or start RTFP.

1.5.13 Utility functions - task operations

Overview

RTFP is based on Amalthea models, therefore, in order to verify the behavior described by the model, without implementing the functiinality is "cycle wasting", Where the processor is used for the decided WCET (Worst Case Execution Time) of the task.

Sleep in Epiphany cores

Each Epiphany core has 2 hardware timers, in order to acheive cycle-accurate sleep times, the timer is used to pause the processor in its curent state for a given amount of clock cycles. The function

```
void sleepTimerMs(int ticks, int taskNum);
```

will acheive sleep times accurate to one tick of 1 ms.

It is important to note that any sleep funciton structured similarly to this function should not have a sleeping time that is larger than the tick period of RTFP. This funciton will devide the required sleep ticks to a series of timed sleep periods of 1 tick each, to allow the tick interrupt to run regularly while the sleep function is being used.

1.5.14 Utility functions - Visualization

Overview

RTFP includes utlities to translate the contents of output buffers of each core as well as the contents of shared memory. This is done to conform the user output from Parallella with the content of the Amalthea model used to construct the system.

Distributed shared memory visulaization

The struct *LabelVisual* is used to while printing the legend of the output table seen when running RTFParallella.

```
typedef struct{
unsigned row;
unsigned col;
unsigned num_visible_labels;
}LabelVisual;
```

where:

- *row* is absolute row of the target core (core to be visualized)
- *col* is absolute column of the target core
- *num_visible_labels* is the total number of labels in core memory that will be shown in the output table.

An array that holds the indices of labels to be shown should be declared and initialized separately. Example:

```
unsigned index_array1[dstr_mem_sec_1_label_count];
```

To avoid any segmentation fault or accessing elemnts outside the array size, declare the size of the index array to be the number of elements in the target core memory section that is required to be shown.

Shared memory visualization

To visualize shared memory, only an array containing the required visible values of a shared memory section needs to be declared and initialized. Example:

```
unsigned index_array_DRAM[shared_mem_section1_label_count];
```

Output

As in the provided examples, all outputs from the host code running on parallella board is being printed into *stderr* buffer. RTFP provides a view utilies to print shared and distributed shared memory values and indicate value changes.

The following sections will outline those functions and their usage.

Output table legend

```
user_config_print_legend(LabelVisual core_config,unsigned array[])
```

Arguments:

- core_config : row and column and number of visible labels for the core to be visualized.
- array : indices of visible labels.

```
user_config_print_legend_auto(unsigned array_length,unsigned array[]);
```

Arguments:

- array_length : number of visible labels for the shared memory section to be visualized.
- array : indices of visible labels.

Output table values

```
user_config_print_values(LabelVisual core_config,unsigned array[],unsigned int values_
↪array[],unsigned int prv_val_array[])
```

Arguments:

- core_config : row and column and number of visible labels for the core to be visualized.
- array : indices of visible labels.
- values_array : holds values of all labels in a shared memory section.

1.5.15 Adapting FreeRTOS to RTFP

Overview

RTFP uses the services from FreeRTOS to switch tasks and provide other utilities to be used in task management. In this file, the adaptation of FreeRTOS to RTFP will be explained, along with the limitations of FreeRTOS.

RTFP task structure in FreeRTOS

Tasks in RTFP follow the read-execution-write semantics. while FreeRTOS tasks do not consider singl instances that start and finish at defined points time. FreeRTOS tasks are infinite loops which are swapped in and out of execution based on the scheduling policy used.

The following adaptation of a generalized FreeRTOS task that takes AmaltheaTask struct as a parameter shows the semantics of read execution write and the determination of instances of the task as will as the way to make a FreeRTOS task periodic

```
void generalizedRTOSTak(AmaltheaTask task){
        TickType_t xLastWakeTime = xTaskGetTickCount();
        for (;;){
                task.cInHandler();
                task.taskHandler();
                task.cOutHandler();
                vTaskDelayUntil( &xLastWakeTime, task.period);
        }
}
```

Creating FreeRTOS tasks for AmaltheaTask objects

Creating tasks in FreeRTOS is done using the xTaskCreate() function. This functions is used by RTFP to Connect the FreeRTOS generalized task described earlier to the AmaltheaTask structure such that it can be scheduled, switched and managed..

```
xTaskCreate(generalizedRTOSTak,
                        "Task",
                        int stack_size,
                        &(*AmaltheaTask Task),
                        int priority,
                        xTaskHandle handle);
```

Arguments:

- stack_size : the size of the local stack of the task. The size of all labels used by the task should be considered when calculating the required stack size. Also, additional stack should be added for the task control block (TCB) of the task.
- AmaltheaTask task : a pointer to the AmaltheaTask struct that will be passed to the generalized FreeRTOS task.
- priority : Priority of the task.
- handle : handle to the FreeRTOS task created, currently not used by RTFP.

1.5.16 Scheduling example

This page describes the example amalthea model implemented in RTFP.

The implementation of this example can be found in *core0_main.c* and *core1_main.c* sorce files.

Two examples for visualizing this model using the host processor are also provided in *armcode.c* and *host_example1.c*.

Overview

The model implemented in this example has the following attributes:

Number of cores	2
Number of tasks	5
Number of event chains	1

The tasks implemented in this core have the following attributes:

Task name	task period	WCET	deadline	core mapping
task5ms0	5 mS	2 mS	5 mS	Core 0
task10ms0	10 mS	3 mS	10 mS	Core 0
task20ms0	20 mS	5 mS	20 mS	Core 0
task10ms1	10 mS	3 mS	10 mS	Core 1
task20ms1	20 mS	7 mS	20 mS	Core 1

The used cores are mapped on Epiphany cores as follows:

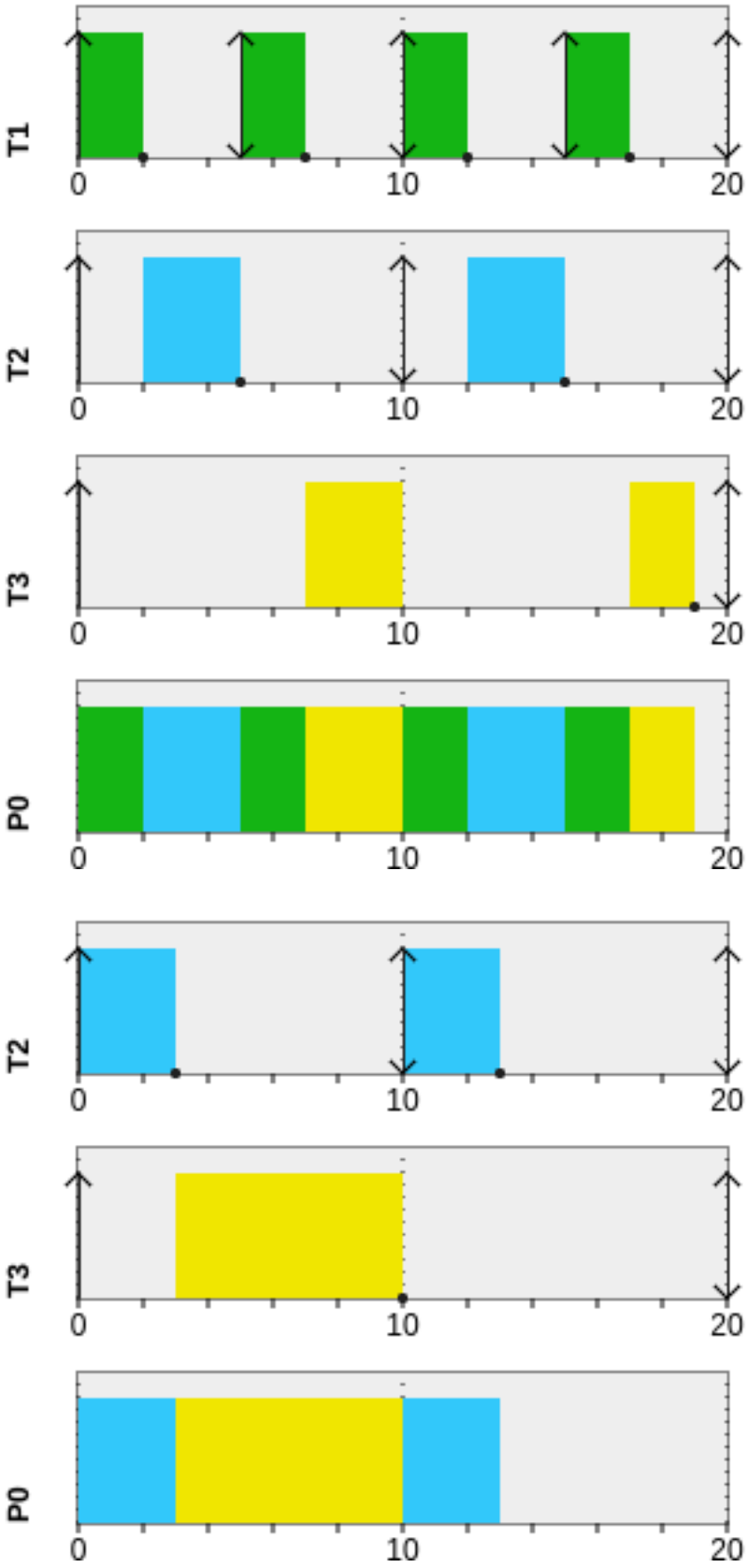
Core name	Epiphany device row	Epiphnay device column
Core0	0	0
Core1	1	0

The model has the following shared memories in separate locations. Each of the shared memory locations holds one memory section that has 10 contiguous labels of size `unsigned int`. The shared memories are shown in the following table:

Shared memory location	Shared memory sections | section label size	
DRAM	1	*unsigned int*
Core0	1	*unsigned int*
Core1	1	*unsigned int*

RMS scheduling of the model

The model has been simulated using the Rate Monotonic (RM) scheduling policy. The simulation has been performed using simso. The gantt charts shown here explain the task scheduling behavior that is desired from any scheduler that follows the RMS policy.


T1
0
10
20
T2
0
10
20
T3
0
10
20
P0
0
10
20
T2
0
10
20
T3
0
10
20
P0
0
10
20

The output from RTFParallella after realizing this behavior sould be as follows:

```
       RFTP demo started
       ================================================================================
     |   Tasks being executed   | observed labels values |
tick |   Core 1    |   Core 2    |
       ================================================================================
   1 |   Task5ms0  |  Task10ms1  |
   1 |   Task5ms0  |  Task10ms1  |
   2 |  Task10ms0  |  Task10ms1  |
   3 |  Task10ms0  |  Task20ms1  |
   4 |  Task10ms0  |  Task20ms1  |
   5 |   Task5ms0  |  Task20ms1  |
   6 |   Task5ms0  |  Task20ms1  |
   6 |   Task5ms0  |  Task20ms1  |
   7 |  Task20ms0  |  Task20ms1  |
   8 |  Task20ms0  |  Task20ms1  |
   9 |  Task20ms0  |  Task20ms1  |
  10 |   Task5ms0  |  Task10ms1  |
  11 |   Task5ms0  |  Task10ms1  |
  12 |  Task10ms0  |  Task10ms1  |
  13 |  Task10ms0  |    no task  |
  14 |  Task10ms0  |    no task  |
  15 |   Task5ms0  |    no task  |
  16 |   Task5ms0  |    no task  |
  17 |  Task20ms0  |    no task  |
  18 |  Task20ms0  |    no task  |
  19 |    no task  |    no task  |
  20 |   Task5ms0  |  Task10ms1  |
```

Example task implementation

Utilities explained earlier are used to realize the tracing of tasks while being executed and switched.

```
void handler10ms(){
        updateDebugFlag(800);
        sleepTimerMs(3,2);
        passes2++;
        traceTaskPasses(2,passes2);
        traceRunningTask(0);
}
```

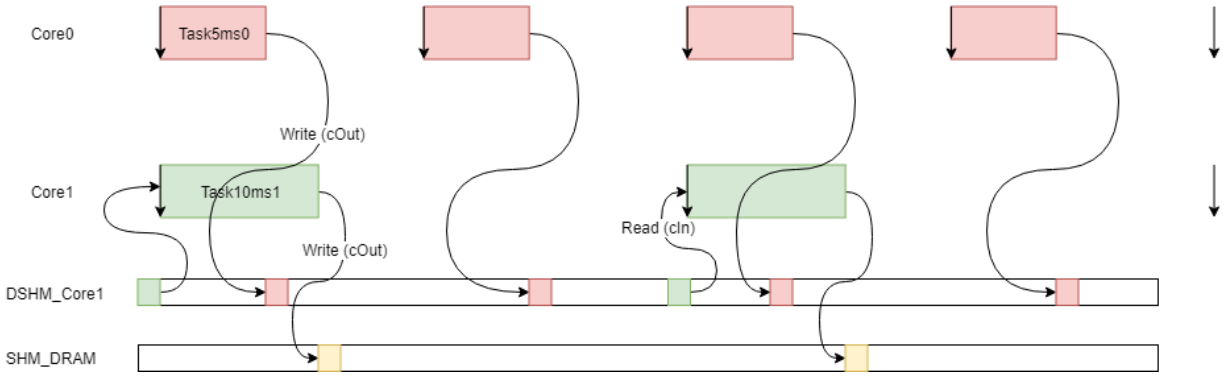
The task is traced within the sleep function while it runs (sleeps if there is no behavior to be performed), and once it completes the sleep period, the job instance of the task will be updated and traced, and the running task flag in output buffer will be voided to 0 to indicate that the core is now idle. Until another task runs and updates the output buffer.

Example event chain implementation

Shared and distributed shared memory management utilities described earlier are used to implement a simple task chain consisting of two tasks, a *producer* and a *consumer*. Namely, the task `task5ms0` being the producer and the `task10ms1` being the consumer task. They share one label in the distributed shared memory of core 1. The output of the event chain is a label in the shared memory of the parallella system.

The input to the event chain is a local label in the producer task that is inceremented by the end of each instanceof the task. This local label simulates data coming from a sensor in an automotive application.

The event chain data propagation is shown in the following figure. Note that the consumer task `task10ms1` is oversampling the producer task.



1.5.17 BTF tracing example

This page describes the example amalthea model implemented in RTFParallella using trace framework.

The implementation of this example can be found in *core0_main.c* and *core1_main.c* source files.

Th example for visualizing this model using the host processor are also provided in *host_example1.c*.

Overview

The model implemented in this example has the following attributes:

Number of cores	2
Number of tasks	5
Number of event chains	1

The tasks implemented in this core have the following attributes:

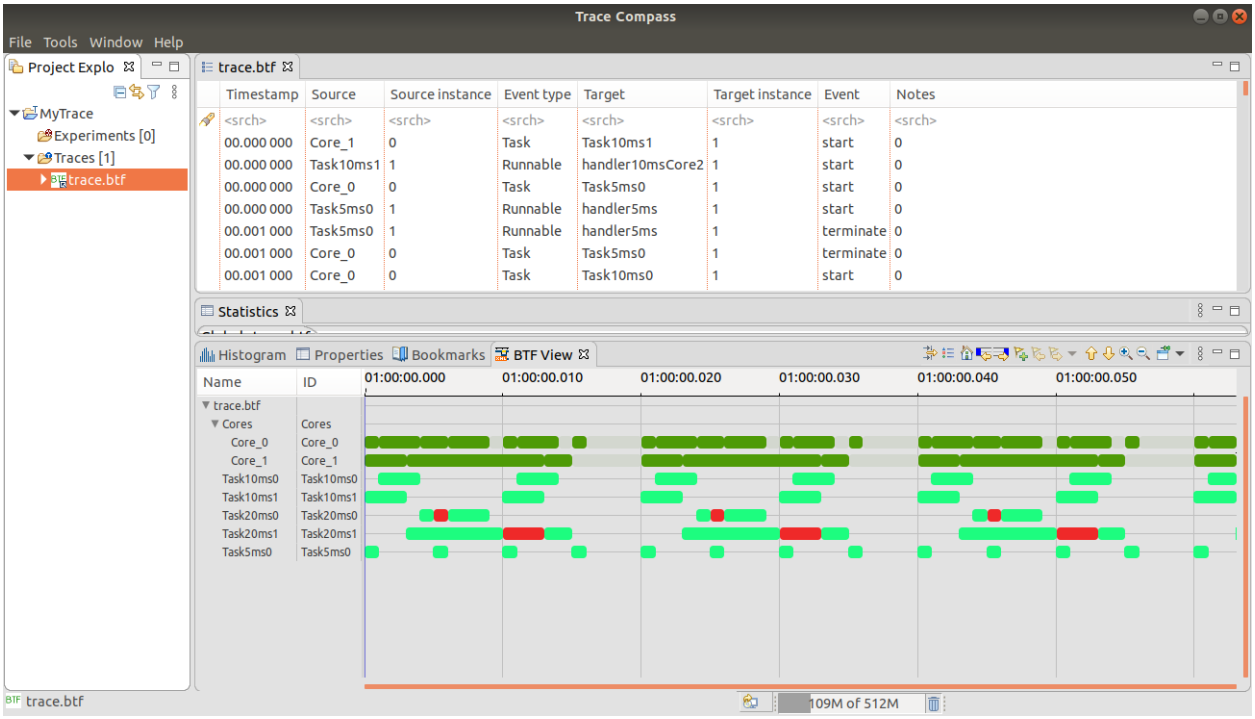
Task name	task period	WCET	deadline	core mapping
task5ms0	5 mS	1 mS	5 mS	Core 0
task10ms0	10 mS	3 mS	10 mS	Core 0
task20ms0	20 mS	4 mS	20 mS	Core 0
task10ms1	10 mS	3 mS	10 mS	Core 1
task20ms1	20 mS	9 mS	20 mS	Core 1

The used cores are mapped on Epiphany cores as follows:

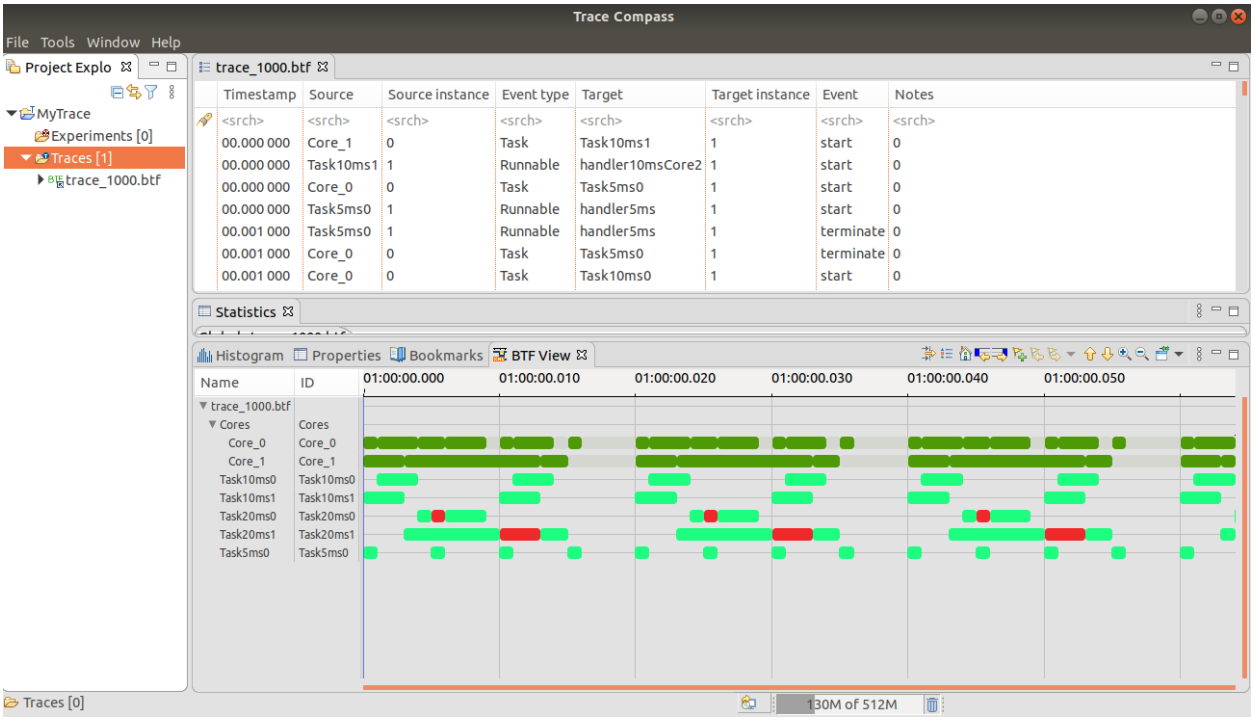
Core name	Epiphany device row	Epiphnay device column
Core0	0	0
Core1	1	0

Results

The above task model is simulated on Parallella hardware platform. The generated BTF trace file can be viewed on *Eclipse Trace Compass*. Below is the resultant trace file viewed on Eclipse Trace Compass with the time scale factor of 100us.



Below is the resultant trace file viewed on Eclipse Trace Compass with the time scale factor of 1000us.



1.5.18 Limitation and Future Work

Limitations

- The current trace framework supports time scale factor of 100 us and 1000 us.

- The tracing framework is restricted to two hardware cores.
- The CdGen application supports code generation for tracing framework only for RMS scheduler.

Future Work

- Support for wider range of time scaling factor.
- Adding the support for more than two hardware cores in the tracing framework.
- Reducing the memory footprint used by each packet of the BTF trace data.
- Adding the tracing framework support for other schedulers like FreeRTOS and POSIX in CdGen tool.

Index

Symbols

D

E

H

I

K

L

P

R

S

T

U

W

X