(GSoC 2019) CPU-GPU Response Time and Mapping Analysis

Release 1.0

Nov 21, 2019

Contents

Google Summer of Code

CHAPTER 1

Motivation

Through one of the subjects in my Master's course, I carried on a project analyzing metrics of Software Models and visualizing it in APP4MC.

It was quite challenging as I was not very familiar with the Amalthea model and its APP4MC platform at first. But soon I was able to understand the concepts and started enjoying it. The project resulted in completing an application delivering performance and reliability metrics of a given Software Model. This is basically my motivation for participating in this Eclipse GSoC project, "CPU-GPU Response Time and Mapping Analysis".

Since the topic's ultimate goal is to achieve systems' real-time determinism for modern HPC (High Performance Computing) applications, analyzing response times is essential and my basic knowledge in regard to, e.g., timing constraints or end-to-end event chain latency values according to the different communication paradigms (direct, implicit, LET: Logical Execution Time) which I obtained through my Master's study were very helpful for me in order apply for and luckily realize this project.

Now that the industry's interest has moved on to "Heterogeneous Systems", I do hope that my GSoC work would be helpful for other researchers in this regard and make a contribution to the further development of the platform.

Ki, Junhyung

CHAPTER 2

Contents

2.1 Intention

The current APP4MC APIs provide several methods for getting execution time for a task, a runnable or ticks (pure computation) through the util package.

However, APIs for response time analysis do not exist yet. The reason why is that response time analysis results highly vary depending on the analyzed model properties such as the scheduling, the mapping, and others.

Since the trends are evolving from homogeneous to heterogeneous platforms, the analysis methodologies have become much more sophisticated. A generic form of CPU response time analysis, which can be used for different mapping models with different types of processing units (e.g., GPU), is though reasonable across modern analysis techniques.

Additionally, this project also aims to offer end-to-end event-chain latency analyses that incorporate a distinct concepts such as reaction & age which will be outlined in this documentation. Such analyses are intended to help users to analyze how much time would be taken for some data to be propagated from the beginning to the end of a given chain of tasks.

2.2 Contribution & Benefits for The Community

In this project, a standardized response time analysis methodology (Mathai Joseph and Paritosh Pandya, 1986) is used. Not only this, but a class, `CPURta` which can be used with various implementations (e.g. a Genetic Mapping Algorithm), is also provided.

Since a heterogeneous platform requires different analysis methodologies for processing units, a class that has a built-in response-time calculation algorithm is very helpful and makes the entire developing circle quicker.

Another class, `RTARuntimeUtil` supports the `CPURta` class by providing several ways to calculate the execution time of a task. The methodology for deriving execution time changes depending on the execution case (e.g., Worst Case, Best Case, Average Case), the offloading mechanism (e.g., Synchronous, Asynchronous), and the mapping model. This class can be modified and reused for other models under analysis simply by adjusting a single method which takes care of memory accessing time (because memory accessing time can be different according to the target hardware).

Furthermore, this GSoC project provides a small GUI implementation, which visually describes the mapping model with information about schedulability, the corresponding response times for each task, and E2E latency analysis results (`E2ELatency`) according to each task chain.

2.3 Milestone with The Goal of Each Phase

2.3.1 Phase 1 (May 27 - June 24)

1. Structuring classes based on the abstraction layers (Top: End-to-End latency / Mid Layer: Response Time / Low Layer: Task & Runnable Execution Time)

Layer	Responsibility
Top	End-to-End Latency
Mid	Task Response Time
Low	Task & Runnable Execution Time

2. Developing task and runnable level execution time methods with taking memory access cost and offloading mechanisms into account
3. Testing
4. Documenting

The main focus of phase 1 is to implement the basis framework and map each and every functionality to the classes. In this way, the entire system becomes organized which eases refactoring and debugging.

2.3.2 Phase 2 (June 25 - July 22)

1. Developing interfaces between classes
2. Implementation of response time analysis algorithms according to different communication paradigms, i.e., direct and implicit communication)
3. Structuring and developing basic user interface class
4. Testing
5. Documenting

The main focus of phase 2 is to provide a stable response time method which can be used for several models under various configuration settings.

2.3.3 Phase 3 (July 23 - August 25)

Refine Previous Phase and E2E Latency Foundation (IC, LET) / Documenting

1. Implementation of E2E latency analysis methodologies according to the concepts such as age, reaction, and propagation under different communication paradigms such as direct, implicit, and LET = Logical Execution Time.
2. Extend and finalize the UI part
3. Testing
4. Final documenting (Through Sphinx & readthedocs)

The main focus of phase 3 is to implement newly defined concepts of end-to-end latency methodologies in line with the implicit and LET communication paradigms. As a consequence, users gain much more task chain metrics in addition to data propagation only.

Moreover, by using the provided GUI, user can investigate mapping scenarios and analyze response times & E2E latency metrics without diving into java implementations.

2.4 Approached Theories

2.4.1 Basic RTA

- Table of Notation for **Basic RTA**

Description	Symbol
Task	i
WC Response time	R_i^+
WC Execution time	C_i^+
Period	T_i
Frequency in Hz	f_m
Latency	L
Read Latency	$L_{\uparrow m \rightarrow l}$
Write Latency	$L_{\downarrow m \rightarrow l}$
Read labels	$\mathcal{R}_i$
Written Labels	$\mathcal{W}_i$
Label	$\mathcal{L}$
Label Size	$\mathcal{S}$

Memory Access Cost

Memory access time is different depending on the target hardware. In this project, the memory access time is defined based on NVIDIA-TX2 platform. The equation for deriving this is referenced the WATERS19 projects namely CPU-GPU Response Time and Mapping Analysis for High-Performance Automotive Systems

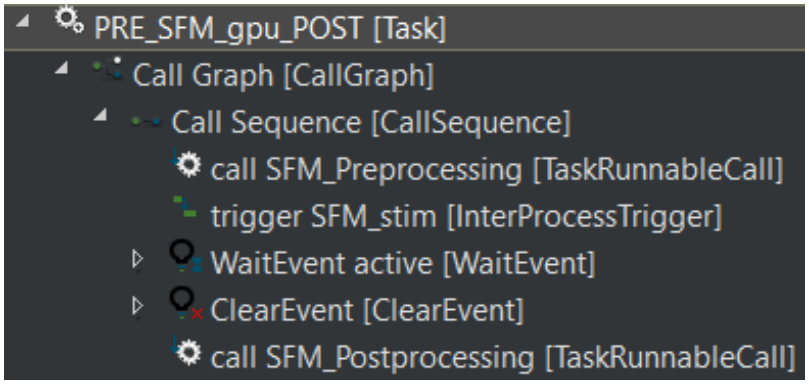
$L^{+}_{a,i} = \sum_{x \in \mathcal{R}_i} \left(\left\lceil \frac{\mathcal{S}_x}{64} \right\rceil \right) \cdot \frac{L_{\uparrow m \to l}}{f_m} + \sum_{y \in \mathcal{W}_i} \left(\left\lceil \frac{\mathcal{S}_y}{64} \right\rceil \right) \cdot \frac{L_{\downarrow m \to l}}{f_m}$

Here, the constant 64 is used as the baseline derived from the WATERS19 challenge description. ls denotes the label size and rl and wl define given read label and write label latencies specified in the given AMALTHEA model.

To find relevant methods, see *CPU Task Execution Time*.

Synchronous & Asynchronous Mechanism

In the provided AMALTHEA WATERS19 model, some of the tasks that are mapped to CPU trigger tasks mapped to GPU. In this case, the execution or response time can be different according to the offloading mechanism.



- **Synchronous**

The triggering task triggers its target GPU task when it reaches `InterProcessTrigger` and actively waits until it receives the triggered task's result after the response from the triggered GPU task. Then it finishes the remaining job.

- **Asynchronous**

The triggering task triggers its target GPU task when it reaches `InterProcessTrigger` and passively waits for the response from the triggered GPU task and finishes the remaining job. During the passive waiting phase, other lower priority tasks can execute on the processor. The asynchronous methodology described here can be modified according to the user's interpretation.

This concept is used in two of the four execution cases introduced by a method, *CPU Task Execution Time*.

Response Time

The response time analysis approach implemented here is not only designed for Multi-core Systems but also for Heterogeneous Systems. Basically, the following classical response time analysis equation is used.

$R^{+}_{i} = C^{+}_{i} + \sum_{j \in hp(i)} \left\lceil \frac{R^{+}_{i-1}}{T_j} \right\rceil C^{+}_{j}$

The equation is based on RMS (Rate Monotonic Scheduling) which means that static priorities are assigned to tasks according to their period. A task with the shorter period results in a higher task priority. Here, R_i denotes the response time of task τ_i and $hp(i)$ is the set of tasks indexes (j) which have a priority higher than task i.

To find relevant methods, see *Response Time Sum*.

2.4.2 End-to-End Latency

The approach and its equations used here are referenced from a yet-unpublished paper, "Model-based Task Chain Latency and Blocking Analysis for Automotive Software" by the same authors who published CPU-GPU Response Time and Mapping Analysis for High-Performance Automotive Systems.

- Table of Notation for **End-to-End Latency**

Symbol	Description
Task	τ
Response time	R
Execution time	C
Period	T
Task chain	γ
Latency	δ
implicit communication	ι
LET communication	λ
Age latency	α
Reaction latency	ρ
Reaction update	υ

Task Chain Reaction

The time between the task chain's first task release to the earliest task response of the last task in the chain.

Task Chain Reaction (Implicit)

- **Best-case Task-Chain Reaction (Implicit Communication Paradigm)**

$\delta^{-}_{\gamma,\rho,\iota} = \sum_j R^{-}_j$ with $\tau_j \in \gamma$

The best-case task chain reaction latency for implicit communication can be calculated by considering the sum of all task's best case response times within task chain. Here, γ refers to a task chain, ρ corresponds the reaction latency, and ι outlines that this latency considers the implicit communication paradigm.

- **Worst-case Task-Chain Reaction (Implicit Communication Paradigm)**

$\delta^{+}_{\gamma,\rho,\iota} = \sum_{j=0}^{j=|\gamma|-2} \left(T_j + R^{+}_j\right) + R^{+}_{j=|\gamma|-1}$ with $\tau_j \in \gamma$

To find relevant methods, see *Task Chain Reaction (Implicit Communication Paradigm)*.

Task Chain Reaction (LET)

- **Best-case Task-Chain Reaction (Logical Execution Time)**

$\delta^-_{\gamma,\rho,\lambda} = \sum_j T_j$ with $\tau_j \in \gamma$

The best-case task chain reaction latency for LET communication is the sum of all task's periods within task chain γ.

- **Worst-case Task-Chain Reaction (Logical Execution Time)**

$\delta^+_{\gamma,\rho,\lambda} = T_{j=0} + \sum_{j=1}^{j=|\gamma|-1} (2 \cdot T_j)$ with $\tau_j \in \gamma$

To find relevant methods, see *Task Chain Reaction (Logical Execution Time Communication Paradigm)*.

Task Chain Age

"The time a task chain result is initially available until the next task chain instance's initial results are available. In other words, the task chain age latency is the maximal time a task chain's results based on the same input persist in memory."

Early Reaction

$\delta^-_{\gamma,\rho_0,\iota} = \sum_{j=0}$

$\delta^+_{\gamma,\rho_0,\iota} = R_{\gamma 0} + \sum_{j=0}^{j=|\gamma|-2} T_{j+1} + \min(T_{j+1}, \epsilon_j + R_{j+1})$

$\epsilon_j = 2 \cdot T_j - R_j - T_{j+1} - \epsilon_{j-1}$ with $\epsilon_{-1} = 0$

To find relevant methods, see *Task Chain Early Reaction*.

- **Worst-case Task-Chain Age (Implicit)**

$\delta^+_{i,\alpha,\iota} = T_{j=0} + \delta^+_{i,\rho_0,\iota} - \delta^-_{i,\rho_0,\iota}$

- **Worst-case Task-Chain Age (LET)**

$\delta^+_{i,\alpha,\lambda} = T_{j=0} + \delta^-_{i,\rho,\lambda} - \delta^+_{i,\rho,\lambda}$

To find relevant methods, see *Task Chain Age*.

Data Age

It describes the longest time some data version persists in memory. This is independent of task chains and simply depends on the period of entities writing a particular label (i.e. data).

- Best-case Task Age

$\delta^-_{i,\alpha} = T_i - R^+_i + R^-_i$

- Worst-case Task Age

$\delta^+_{i,\alpha} = T_i - R^-_i + R^+_i$

- **Best-case Data Age**

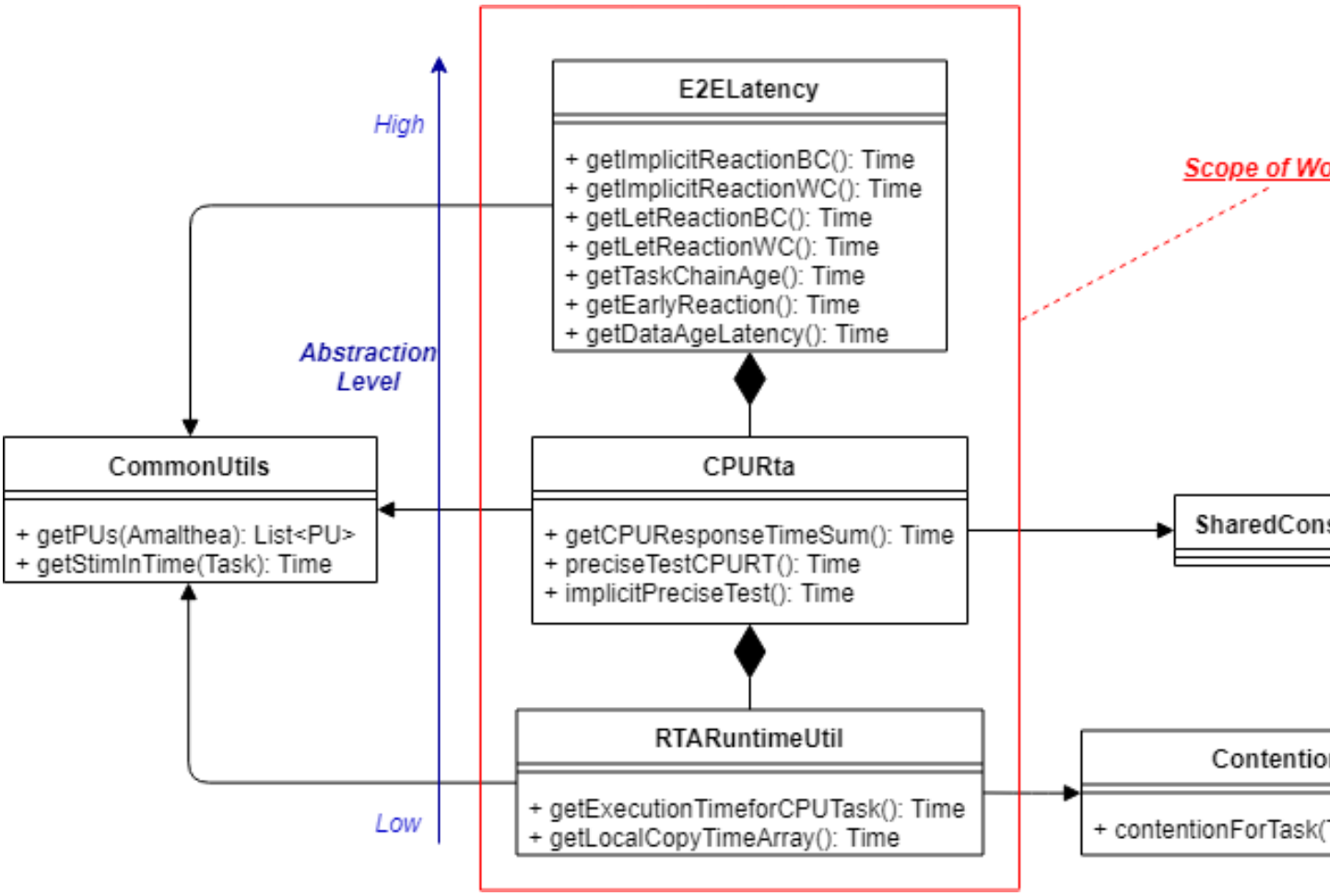
$\delta^-_{l,\alpha} = \min_i \delta^-_{i,\alpha}$ with τ_i being any task that accesses label l.

- **Worst-case Data Age**

$\delta^+_{l,\alpha} = \min_i \delta^+_{i,\alpha}$ with τ_i being any task that accesses label l.

To find relevant methods, see *Data Age*.

2.5 Class Tree with Implemented Methods



The above UML class diagram describes the project's implementation in a hierarchical way.

2.5.1 Key Classes

E2ELatency

The top layer takes care of the end-to-end latency calculation of the observed task-chain based on the analyzed response time from the CPURta class. It includes calculating E2E latency values according to the concepts stated in the theory part (e.g., Reaction, Age).

Task Chain Reaction (Implicit Communication Paradigm)

```
public Time getTCReactionBC(final EventChain ec, final ComParadigm paradigm, final␣
↪CPURta cpurta)
```

This method derives the given event-chain's best-case end-to-end latency based on the reaction concept for the direct and implicit communication paradigms.

Code Reference

```
public Time getTCReactionWC(final EventChain ec, final ComParadigm paradigm, final␣
↪CPURta cpurta)
```

This method derives the given event-chain's worst-case end-to-end latency value based on the reaction concept for the direct and implicit communication paradigms.

Code Reference

For the details, see *Task Chain Reaction (Implicit)* and *UI Features (E2ELatency)*.

Task Chain Reaction (Logical Execution Time Communication Paradigm)

```
public Time getLetReactionBC(final EventChain ec, final CPURta cpurta)
```

This method derives the given event-chain's best-case end-to-end latency value based on the reaction concept for LET communication.

Code Reference

```
public Time getLetReactionWC(final EventChain ec, final CPURta cpurta)
```

This method derives the given event-chain's worst-case end-to-end latency based on the reaction concept for LET communication.

Code Reference

For the details, see *Task Chain Reaction (LET)* and *UI Features (E2ELatency)*.

Task Chain Age

```
public Time getTaskChainAge(final EventChain ec, final TimeType executionCase, final␣
↪ComParadigm paradigm, final CPURta cpurta)
```

This method derives the given event-chain latency based on the age concept. By changing `TimeType executionCase` parameter, the latency in the best-case or the worst-case can be derived.

Code Reference

For the details, see *Task Chain Age* and *UI Features (E2ELatency)*.

Task Chain Early Reaction

```
public Time getEarlyReaction(final EventChain ec, final TimeType executionCase, final␣
↪ComParadigm paradigm, final CPURta cpurta)
```

This is a method to be pre-executed for getting the reaction-update latency values. The best-case and worst-case early-reaction latency values should be derived first and then the reaction update latency can be calculated. By changing `TimeType executionCase` parameter, the latency in the best-case or the worst-case can be derived.

Code Reference

For the details, see *Early Reaction* and *UI Features (E2ELatency)*.

Data Age

```
public Time getDataAge(final Label label, final EventChain ec, final TimeType␣
↪executionCase, final ComParadigm paradigm, final CPURta cpurta)
```

This method derives the given label's age latency. If the passed event-chain does not contain the observed label, `null` is returned. By changing `TimeType executionCase` parameter, the latency in the best-case or the worst-case can be derived.

Code Reference

For the details, see *Data Age* and *UI Features (E2ELatency)*.

CPURta

The middle layer takes care of analyzing task response times. It is responsible for calculating response times according to the communication paradigm (Direct or Implicit communication paradigm).

Response Time Sum

```
public Time getCPUResponseTimeSum(final TimeType executionCase)
```

This method derives the sum of all the tasks' response times according to the given mapping model (which is described as an integer array). The method can be used as a metric to assess a mapping model.

Code Reference

Response Time (Direct Communication Paradigm)

```
public Time preciseTestCPURT(final Task task, final List<Task> taskList, final␣
↪TimeType executionCase, final ProcessingUnit pu)
```

This method derives the response time of the observed task according to the classic response time equation. The response time can be different depending on the passed taskList which is derived from the mapping model. Here, we are concerning response time for RMS (Rate Monotonic Scheduling). It means that a task with the shorter period obtains a higher priority. Before the taskList is passed to the method, it should be sorted in the order of shortest to longest and this job is done by `taskSorting(List<Task> taskList)` which is a private method.

Code Reference

Response Time (Implicit Communication Paradigm)

```
public Time implicitPreciseTest(final Task task, final List<Task> taskList, final␣
↪TimeType executionCase, final ProcessingUnit pu, final CPURta cpurta)
```

This method derives the response time of the task parameter according to the classic response time equation but in the implicit communication paradigm. In the implicit communication paradigm which is introduced by AUTOSAR. A task copies in its required data (labels) to its local memory at the beginning of its execution, computes in the local memory and finally copies out the result to the shared memory. Due to these copy-in & copy-out costs, extra time must be added to the task's execution time which is done by `getLocalCopyTimeArray` (for the details, see *Local Copy Cost for the Implicit Communication Paradigm*) which is a method from the `RTARuntimeUtil` class. As a result, the task's execution time gets longer while its period should stays the same. Once the local-copy cost is taken into account, the remaining process is the same as *Response Time (Direct Communication Paradigm)*

Code Reference

For the details, see *Response Time* and *UI Features (RTA)*.

RTARuntimeUtil

The bottom layer takes care of task and runnable execution time. It is responsible for calculating memory access costs, execution ticks or execution needs, and computation time.

CPU Task Execution Time

```
public Time getExecutionTimeforCPUTask(final Task task, final ProcessingUnit pu,␣
↪final TimeType executionCase, final CPURta cpurta)
```

This method derives the execution time of the task parameter under one of the following cases:

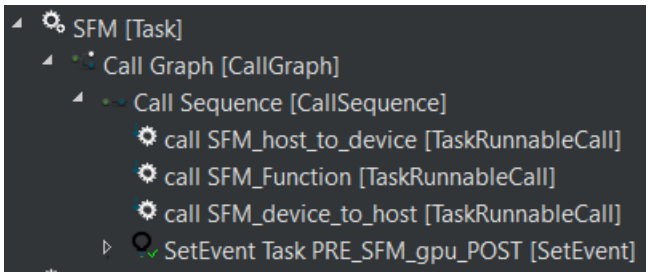
- The CPU task triggers a GPU task in the synchronous offloading mode
- The CPU task triggers a GPU task in the asynchronous offloading mode

(For the details, see *Synchronous & Asynchronous Mechanism*.)

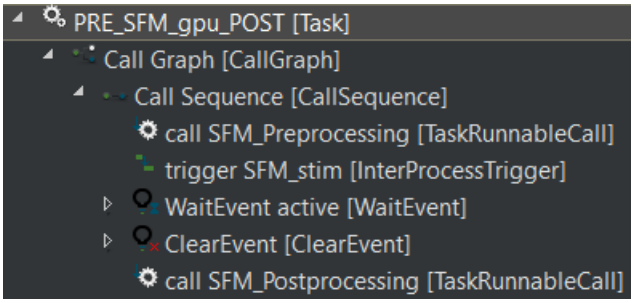
- The GPU task is mapped to a CPU

According to the WATERS challenge, a triggering task (`PRE_..._POST`) can be ignored if the triggered task is mapped to a CPU.

For example, the following Figure shows the `SFM` task which is mapped to the GPU by default.



If the task is mapped to CPU, the offloading runnables (`SFM_host_to_device`, `SFM_device_to_host`) which are in charge of offloading workload to GPU and copying back to CPU are obsolete.



Instead, the labels from runnables before (`Pre-processing`) & after (`Post-processing`) the `InterProcessTrigger` are considered. For the runnable, `Pre-processing`, read labels and read latency values are taken into account. For the runnable, `Post-processing`, write labels and write latency values are taken into account. This job is done by the private method `getExecutionTimeForGPUTaskOnCPU()`.

- Task with only Ticks (pure computation)

When a CPU task without any triggering behavior is passed, only the execution time that corresponds to the task's ticks is considered.

Code Reference for getExecutionTimeforCPUTask

Except for the very last case (Task with only Ticks), the task execution time calculation always includes memory accessing costs. Calculating memory accessing costs is taken care of by methods such as `getExecutionTimeForRTARunnable`, `getRunnableMemoryAccessTime` which are defined as private.

Code Reference for getExecutionTimeForRTARunnable Code Reference for getRunnableMemoryAccessTime

For the details, see *Memory Access Cost*.

Local Copy Cost for the Implicit Communication Paradigm

```
public Time[] getLocalCopyTimeArray(final Task task, final ProcessingUnit pu, final␣
↪TimeType executionCase, final CPURta cpurta)
```

As it is introduced in *Response Time (Implicit Communication Paradigm)*, label copy-in and copy-out costs should be calculated and added to the total execution time of the target task.

The following equation from End-To-End Latency Characterization of Implicit and LET Communication Models is used to calculate these costs.

$C_i^0 = \sum_{l \in I_i} \xi_l(x)$

Where C_i^0 denotes the execution time of the runnable `tau_0`, I_i represents the inputs (read labels) of the considered task and $\xi_l(x)$ denotes the time it takes to access a shared label l from memory x.

$C_i^last = \sum_{l \in O_i} \xi_l(x)$

Where C_i^last denotes the execution time of the runnable `tau_last`, O_i represents the outputs (write labels) of the considered task and $\xi_l(x)$ denotes the time it takes to access a shared label l from memory x.

For the copy-in cost, only read labels should be taken into account. The copy-in cost time is stored on index 0 of the return array. This will later be considered as the execution time of the copy-in runnable which is added to the beginning of the task execution.

For the copy-in cost, only write labels should be taken into account. The copy-in cost time is stored on index 1 of the return array. This will later be considered as the execution time of the copy-out runnable which is added to the end of the task execution.

Code Reference

2.5.2 Supplementary Classes (Out of scope)

SharedConsts

This class is in charge of setting configuration variables. The user can set the offloading mechanism and the execution case (WC, AC, BC) by changing `synchronousOffloading` and `timeType` respectively. Also, all file paths for every Amalthea model can be saved as `String` type constants here so that the user can change the target Amalthea model by switching these constants.

CommonUtils

```
public static List<ProcessingUnit> getPUs(final Amalthea amalthea)
```

This method derives a list of processing units of the target `Amalthea` model. It places CPU type processing units in the front and that of GPU type in the tail (end) of the list.

Code Reference

```
public static Time getStimInTime(final Task t)
```

This method returns the periodic recurrence time of the target task. If the passed task is not a periodic task (e.g., GPU task), the recurrence time of a task which is periodic and triggers the target task is returned. Otherwise time 0 is returned.

Code Reference

Contention

```
public Time contentionForTask(final Task task)
```

This method derives a memory contention time which represents the delay when more than one CPU core and/or the GPU is accessing memory at the same time.

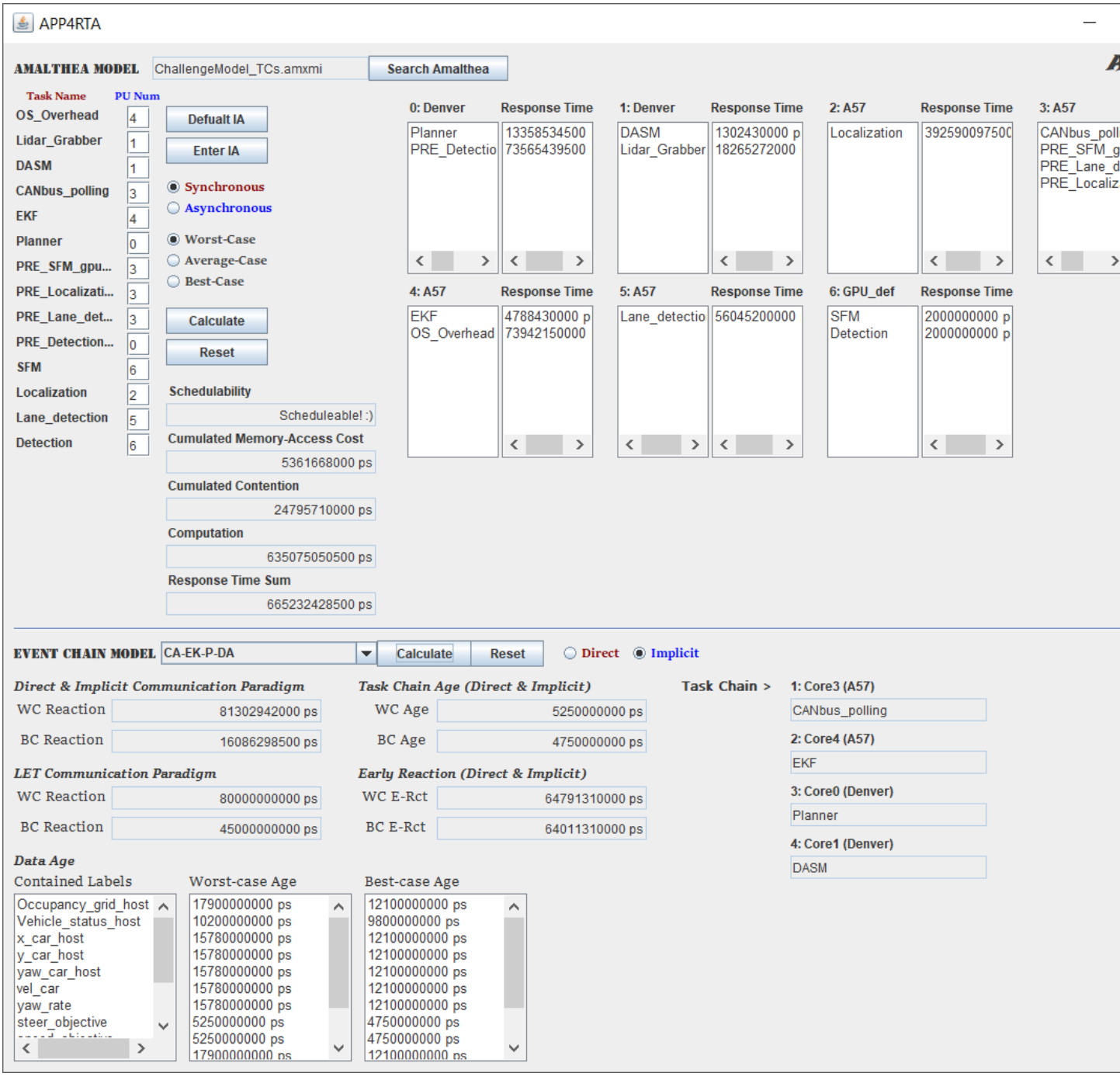
Code Reference

For the details, see Memory Contention Model.

2.6 User Interface (APP4RTA)

2.6.1 Set Up

For analyzing response time & end-to-end event-chain latency

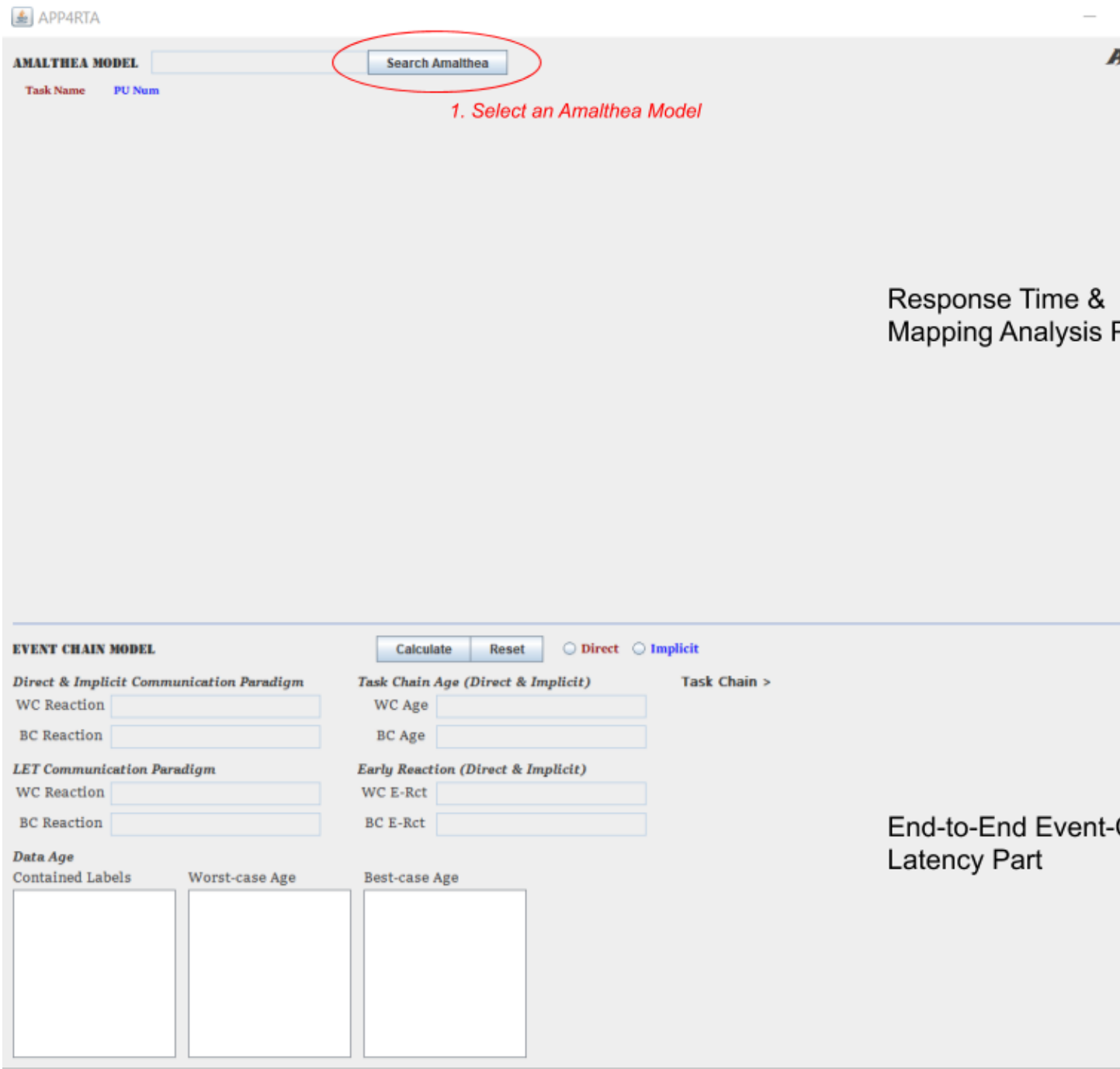


Before executing the code, please install the Java GUI softwares.

- To install Java GUI softwares:

1. Eclipse > `Help`
2. `Install New Software` > Work with: Eclipse Repository (http://download.eclipse.org/releases/oxygen)
3. `General Purpose Tools` > all click from `Swing Designer` to `WindowBuilder XML Core (requires Eclipse WTP/WST)`
4. `Next` > `Next` > `accept` > `Finish`

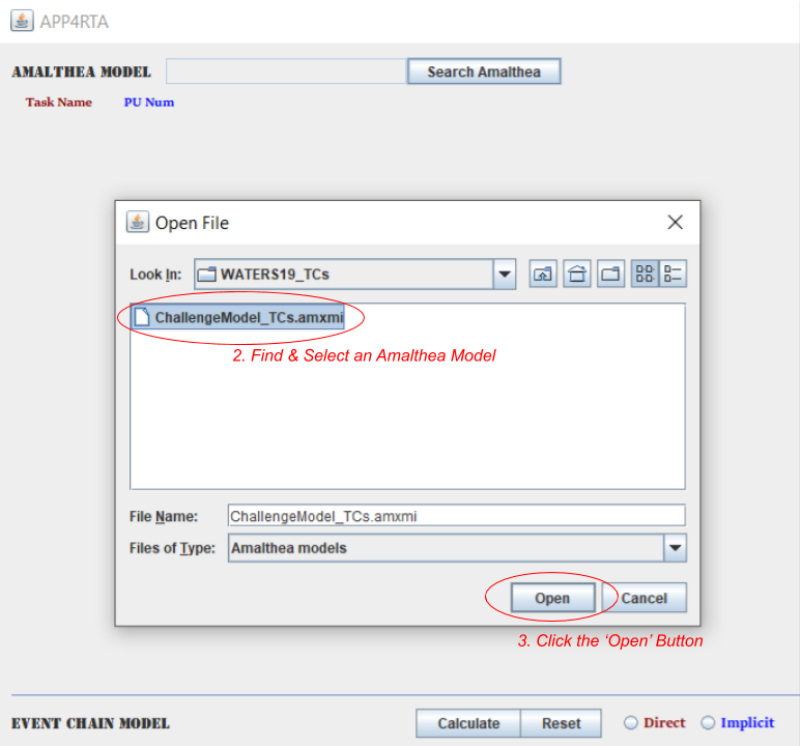
2.6.2 Search Amalthea



Run `APP4RTA.java` in `org.eclipse.app4mc.gsoc_rta.ui` package, then this window will show up. Based on the horizontal line on the middle, the upper part is for response time & mapping analysis, and the lower part is for end-to-end event-chain latency analysis. The first thing to do is deciding a target Amalthea model.

1. The window browser for searching Amalthea models shows up when the `Search Amalthea` button clicked.

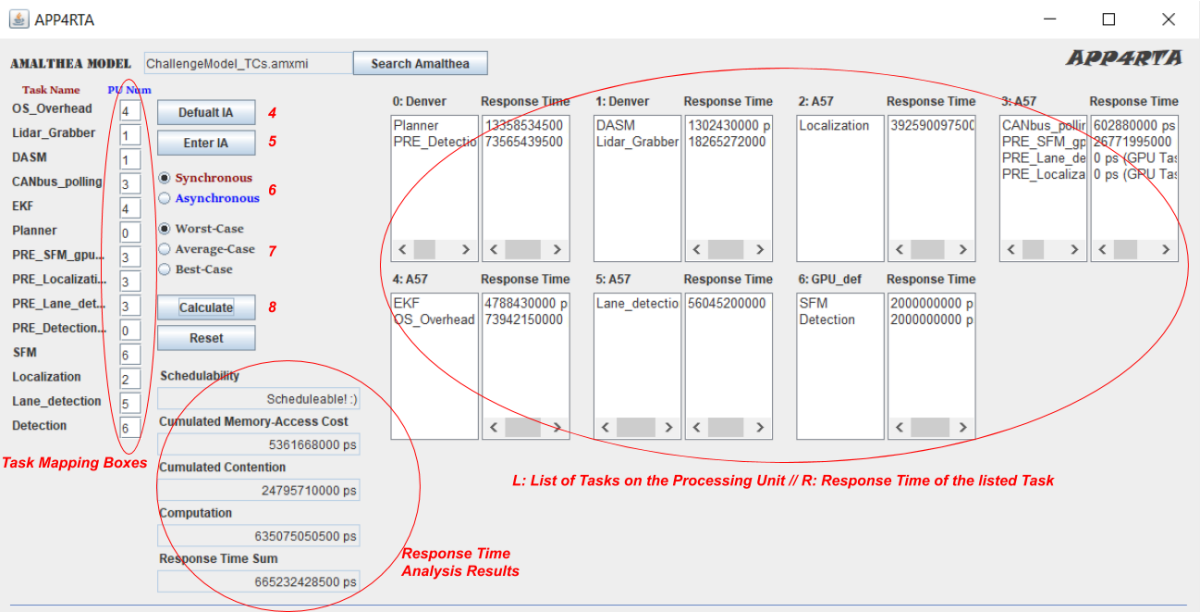
2.6.3 Direct & Select Amalthea



2. When the search browser shows up, direct to the path where the target Amalthea model file is located and select the model file.

3. Click the `Open` button.

2.6.4 UI Features (RTA)



Then the empty space will be filled with the the tasks and processing units of the selected model. On the left-hand side, tasks' names with empty boxes can be found. On the right-hand side, seven pairs of lists are seen (It means the selected model has seven processing units). The list on the left side of each pair is for listing names of the tasks which are mapped to the corresponding processing unit while one on the right side is for listing response times of the corresponding tasks. Basically, we can map the tasks with these boxes by entering the number of each processing unit which is stated on the top of the lists on the left-side.

4. The user can either manually type numbers for every box or simply click the `Default IA` button which would automatically fill up every box with the pre-defined integer array values.

5. Once every `PU Num` box is filled, click `Enter IA` button to assign tasks to processing units according to each integer value. Once this is done, the mapped tasks would appear on the left-side lists.

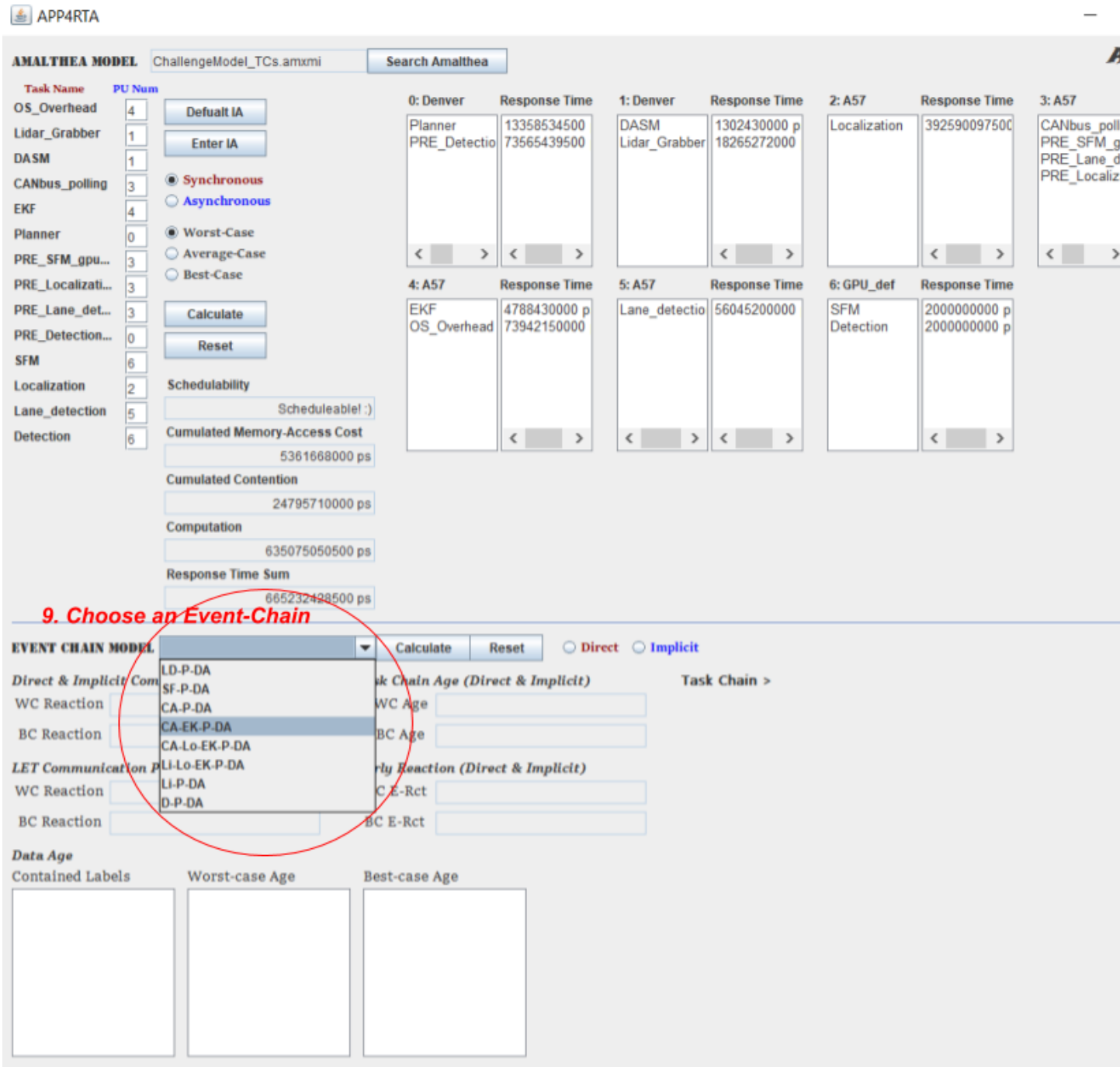
6. Choose the offloading mode between `Synchronous` case and `Asynchronous` case.

7. Choose the execution case between `Worst` case and `Average` case and `Best` case.

8. By clicking the `Calculate` button, all calculation results will be printed out on the text-fields (`Schedulability`, `Cumulated Memory-Access Cost`, `Cumulated Contention`, `Computation`).

For the implementation details, see CPURta-reference.

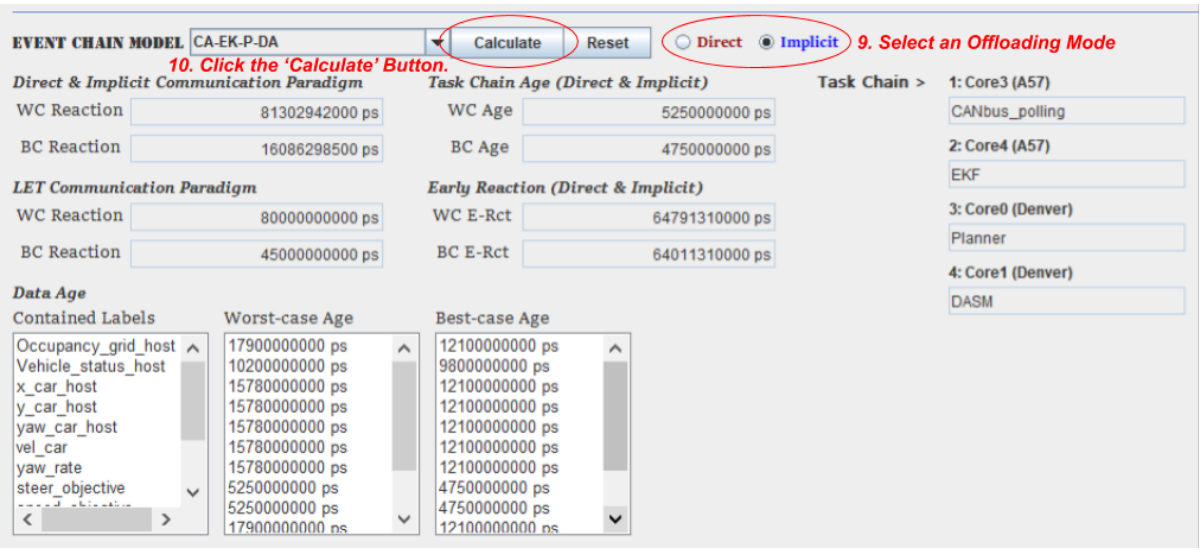
2.6.5 Select an Event-Chain



The event-chain combo-box becomes visible once the user clicks `Enter IA` to assign tasks to processing units according to each integer value in the boxes.

9. To analyze end-to-end event-chain latency, an event-chain in the combo-box should be selected first.

2.6.6 UI Features (E2ELatency)



10. Select the communication paradigm between direct Communication and implicit communication.

11. Finally, click the `Calculate` button.

Then all calculation results regarding reaction, age of data, task-chain in the worst and best cases will be printed out to the corresponding text fields or lists.

For the implementation details, see *E2ELatency*.

Download `PDF` file to see offline.

2.7 Future Work

Many implementations and tests have been left for the future due to the limited time but the topic has so much potential to be developed further. The future work concerns the followings:

2.7.1 1. Reaction Update

The current implementation covers `Early Reaction` but does not cover `Reaction Update`. To calculate `Reaction Update`, the number of sampled task-chain entity instances should be taken into account first, and then `Early Reaction` can finally be utilized to get `Reaction Update`. For the details, see reaction-update.

2.7.2 2. Blocking

The current implementation only focuses on preemtive tasks but does not cover cooperative tasks. Preemptive tasks preempt each other at any moment in time while cooperative tasks preempt each other at runnable boundaries. Therefore, the preempting task should be blocked until the currently executing runnable of the preempted task to finish.

2.7.3 3. Scheduling mode: EDF

The type of real-time scheduling algorithm used in this project is RMS (Rate Monotonic Scheduling). Under RMS, a task with the shorter period obtains a higher priority. To analyze different response times and mapping scenarios, extending the current scheduling algorithm further to EDF (Earliest Deadline First) can be done. Under EDF, tasks are sorted by using their deadlines. Therefore, a task which has the earliest deadline runs first.

2.7.4 4. Read & Write latency setting feature

The current implementation derives `memory access costs` with `read` and `write` latency attribute values from the processing unit. If the selected model does not describe these attributes, the default latency value is assigned to the processing unit and then the `memory access costs` is calculated. Therefore, having a GUI feature for assigning these attribute values is reasonable and useful for users to analyze with different processing unit configurations.

2.7.5 5. Data Age metrics should be organized

Currently, the GUI features for `Data Age` latency are not well-designed because the list for label names and the rest of the lists for latency values are not synchronized. Therefore, this should be restructured in a more tidy way to prevent possible confusions.

With these extensions, `APP4RTA` users can analyze response times under more various configuration settings with the better quality of GUI features.

2.8 Repositories

2.8.1 Eclipse Contribution Tagged Repo

Click Eclipse Contribution Tagged Repository

2.8.2 ReadTheDocs Repo

Click ReadTheDocs Documentation Repository

2.9 Reference

[1] Finding Response Times in a Real-Time System (Mathai Joseph and Paritosh Pandya, 1986)

[2] End-To-End Latency Characterization of Implicit and LET Communication Models (Jorge Martinez, Ignacio Sanudo, Paolo Burgio and Marko Bertogna, 2017)

[3] CPU-GPU Response Time and Mapping Analysis for High-Performance Automotive Systems (Robert Höttger, Junhyung Ki, The Bao Bui, Burkhard Igel and Olaf Spinczyk, 2019)

[4] Model-based Task Chain Latency and Blocking Analysis for Automotive Software (Robert Höttger, The Bao Bui, Junhyung Ki, Burkhard Igel and Olaf Spinczyk, Not yet published)

2.10 Contact

Name: Junhyung Ki

Personal Email: kijoonh91@gmail.com

Student Email: junhyung.ki001@stud.fh-dortmund.de

LinkedIn