A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimisation

D2.2 First release of the meta parallel programming abstraction and the single-criterion performance-aware

Version 1.0

Documentation Information

Contract Number	871669
Project Website	www.ampere-euproject.eu
Contratual Deadline	30.03.2021
Dissemination Level	[PU]
Nature	DEM
Author	Sara Royuela (BSC)
Contributors	Björn Forsberg (ETHZ) Tiago Carvalho (ISEP) Alexandre Amory (SSSA) Delphine Longuet (TRT)
Reviewer	Luis Miguel Pinho (ISEP)
Keywords	Code synthesis, performance, task dependency graph,correct-by-construction

The AMPERE project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871669.

Change Log

Version	Description Change
V0.1	Initial version by BSC
V0.2	Contributions on DSML constraints by TRT
V0.3	Contributions on DSML constraints by SSSA
V0.4	Contributions on Timing by ISEP
V0.5	Contributions on Energy by ETHZ
V0.6	Review by ISEP
V1.0	Final review addressing comments by BSC

Table of Contents

Executive summary

This deliverable covers the work done during the second phase of the project within WP2. The deliverable spans 8 months' work, as defined in the Grant Agreement [1] (from month 8 until month 15, including information that was presented in the first Technical Review, but not in D2.1 [2]), and includes the work done in *Task 2.2*, *Meta parallel programming abstraction and parallel programming model extensions* and *Task 2.3*, *Performance-aware transformation techniques* for the defined period of time, to reach milestone 2 (MS2).

Concretely, the deliverable covers the activities conducted within WP2 towards the implementation of a code synthesis component capable of generating the optimized parallel code based on the requirements specified in the DSML and the information gathered by the tools for multi-criterion analysis. For this purpose, Task 2.3 studies automatic parallelization techniques, via a code synthesis tool, to transform the meta model-driven abstraction (described in D1.3 [3]) into the meta parallel programming abstraction (described in this deliverable). Furthermore, Task 2.2 defines the meta parallel programming model interface, which gathers the information exposed in the meta model-driven abstraction, and also the results of the analysis of the multi-criterion optimization layer.

The targets at MS2 are: (1) a first release of the meta parallel model abstraction upon which model transformations can be applied, and (2) an initial set of model transformation methods targeting only parallel performance. The first milestones of Tasks 2.2 and 2.3 have been carried out successfully, and all objectives of MS2 have been reached and documented in this deliverable. Furthermore, we have extended the work expected in this deliverable to tackle the correctness of the techniques developed in Task 2.3 towards automatic parallelization. The main motivation for this extension is to maintain the *correct-by-construction* paradigm of the model-driven engineering (MDE) systems we target in AMPERE, and thus provide some safety guarantees with respect to the model.

To show the pipeline implemented for AMPERE towards increasing the performance of CPSs we have recorded a demonstration in a video and made it available at B2DROP repository of BSC through the following link: `https://b2drop.bsc.es/index.php/s/yPGEn2wjGrAddeN` and by using the password: $RRAXqgZ3$.

1 Introduction

WP2 aims to develop a meta parallel programming abstraction independent of the underlying processor architecture, capable of capturing all system functional and non-functional requirements, as well as incorporating the parallel semantics required to enable an efficient model transformation, optimized for performance, timing, resiliency, cyber-security and energy-efficiency. Figure 1 depicts the AMPERE software architecture, with the different components and the communication between them. WP2 is enclosed in the greenish box in the middle. This deliverable, including tasks T2.2 and T2.3, corresponds to the dotted green box containing the meta parallel programming model abstraction and the transformation into a high-level parallel programming model, particularly OpenMP.

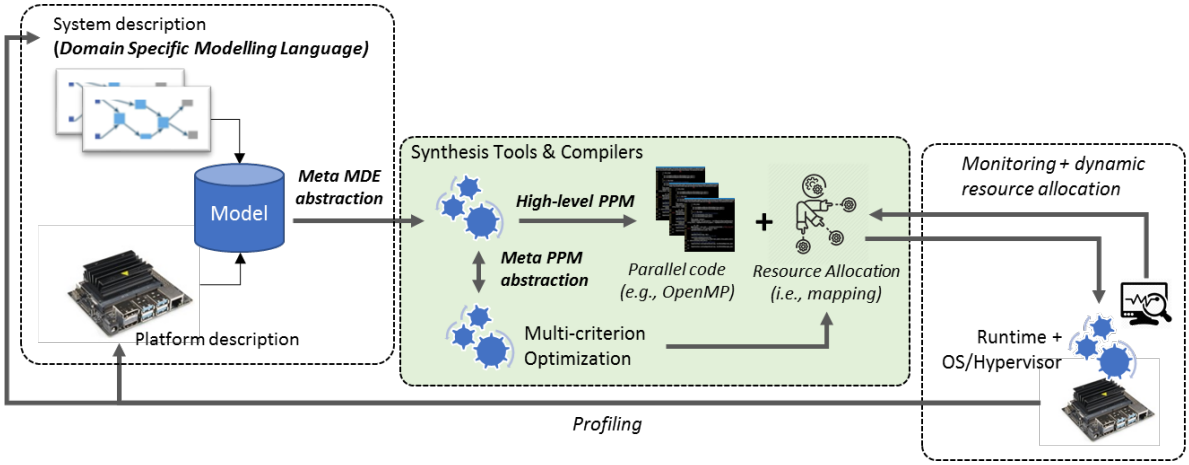


Figure 1: AMPERE's software architecture.

More specifically, Figure 2 shows the detailed pipeline communicating the software components included in the green box part of the AMPERE software architecture from Figure 1. The components involved in this deliverable are highlighted in green, and the interfaces communicating them are outlined in the edges (further details in deliverable D6.2 [4]).

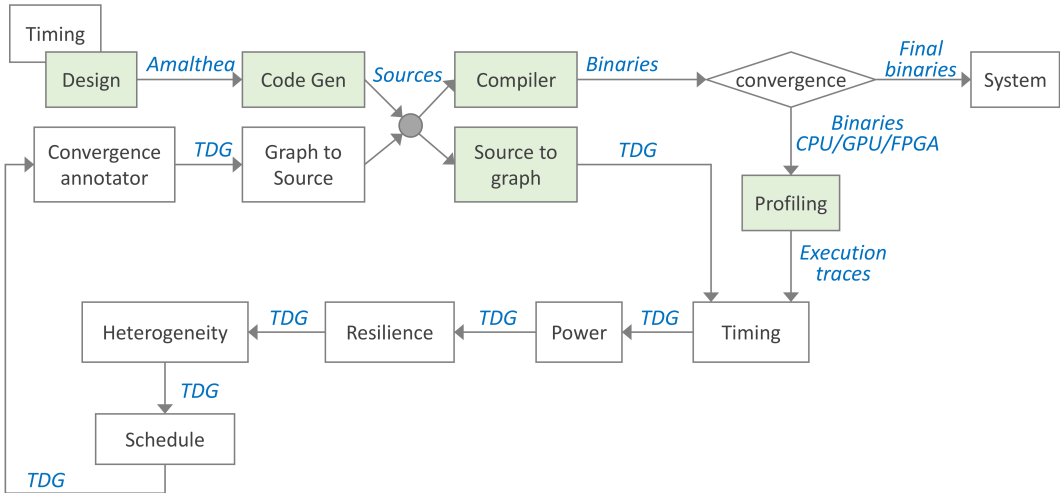


Figure 2: AMPERE's offline components pipeline.

Figure 3 shows the synergies between the two tasks contributing to this deliverable (i.e., T2.2 and T2.3) and those tasks from others WPs that are also due by MS2[1]. Furthermore, Table 1 relates the tasks mentioned in Figure 3 together with the deliverables they will be included in.

[1]Since both T2.2 and T2.3 span more than this deliverable, there are synergies that are not represented here and will appear in D2.3

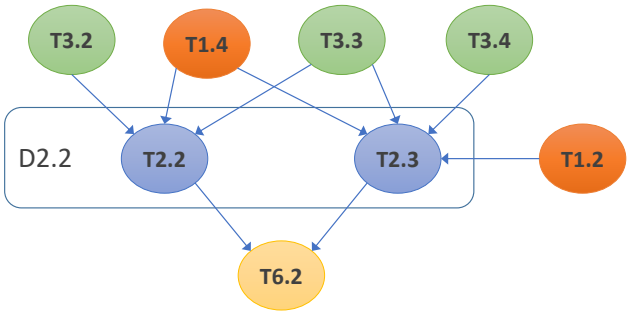


Figure 3: Synergies between T2.2 and T2.3, from WP2, and tasks in other WPs.

Table 1: Tasks and deliverables related to D2.2

Deliverable	D. leader	Task	T. leader
D1.2. Analysis of functional safety aspects on single-criterion optimisation and first release of the test bench suite	BOS	T1.2. Generation of AMPERE test bench suite and use case preparation	BOS
D1.3. First release of the meta model-driven abstraction	SSSA	T1.4. Meta model-driven abstraction and model-driven extensions	SSSA
D3.2. Single-criterion energy optimisation framework, predictable execution models and software resilient techniques	ETHZ	T3.2. Energy optimisation strategies T3.3. Predictable execution models T3.4. Resilient software techniques	BSC
D6.3. Single-criterion AMPERE ecosystem	THALES	T6.2. Synthesis tool integration	BSC

This deliverable is a continuation of *D2.1, Model transformation requirements* [2]. In that deliverable, we presented (a) an analysis of different model-driven engineering (MDE) technologies, including AMALTHEA, AUTOSAR and CAPELLA, and (b) an analysis of different parallel programming models (PPM), including OpenMP, OmpSs, OpenCL and OpenMP. As a result of these analysis, and given the requirements of the project on the DSML level [5], the AMPERE project will use CAPELLA for high-level modeling (particularly of the Obstacle Detection Avoidance System -ODAS- use case), and AMALTHEA [6] to describe the low level details (e.g., requirements for performance) of the system at the DSML level, and OpenMP [7] to orchestrate parallelism in the host, and between the host and the accelerator devices.

To reach the goals of WP2, this deliverable continues the work started in D2.1, and contributes as follows: Chapter 2 describes the parallelism exposed in the vanilla version of AMALTHEA, and introduces the extensions proposed in the model for describing parallelism; Chapter 3 describes the translation of an AMALTHEA model exploiting inter-runnable parallelism into OpenMP via the enhanced AMALTHEA code synthesis tool; Chapter 4 provides a high-level description of the meta parallel programming abstraction (further defined in D6.2 [4]); Chapter 5 describes the benefits of using a code synthesis tool together with HPC correctness techniques to maintain the *correct-by-construction* paradigm of the MDE technologies; Chapter 6 shows a preliminary evaluation of the proposed framework for parallelism in terms of programmability, portability and performance; and Chapter 7 provides the conclusions extracted from the work done for this deliverable.

(Programming model extensions and the multi criteria performance-aware component).

2 Parallelism in AMALTHEA

Based on how components are described with AMALTHEA, this standard enables three different levels of granularity in which parallelism can potentially be exploited:

1. *Among tasks:* this is the only option currently feasible, and it is done through the OS scheduler. It provides a set of synchronization mechanisms among tasks to ensure the correct order of execution.
2. *Among runnables:* this option is currently not supported because the models force runnables to execute sequentially within a task.
3. *Inside runnables:* this option is transparent to the MDE standards because the internals of the runnables are not exposed to the model, and hence it is not visible to the OS scheduler.

Based on our analysis, we concluded that [2]:

1. *Parallelism among tasks* is a coarse-grained level of parallelism suitable for being exploited by the OS, as it is now.
2. *Parallelism among runnables* is a fine-grained level of parallelism suitable for being exploited by the high-level parallel programming model, which can orchestrate the parallel execution in a potentially heterogeneous (host + accelerator) system.
3. *Parallelism inside runnables* is an even finer-grained level of parallelism suitable for being exploited by a parallel programming model (not necessarily the same) in either the host or a dedicated accelerator (e.g., GPU or FPGA).

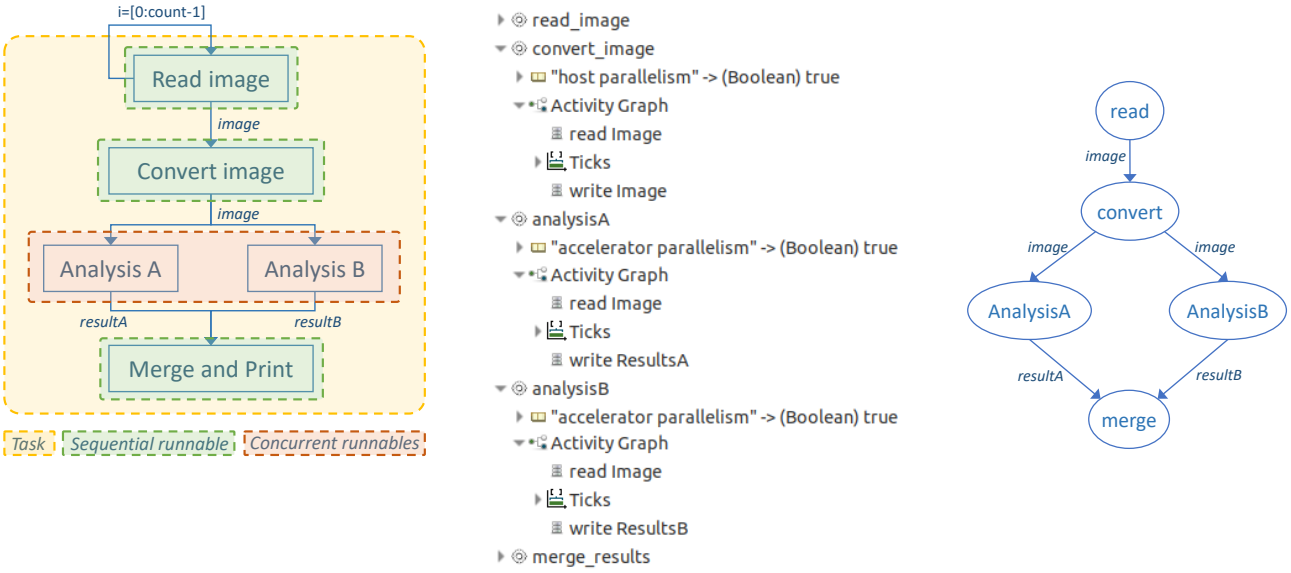
Currently, AMALTHEA only supports the use tasks as units of parallelism. Therefore *the only way to run concurrent runnables is dividing them into different tasks*. Then, these newly created tasks must be properly synchronized with the rest of the tasks, as well as being assigned to a specific scheduler to ensure the correct order of execution. This forces inconveniently the programmer to do the additional exercise of including certain runnables into artificial tasks and relate these tasks by means of stimuli, i.e., *event-based model* instead of data accesses, i.e., *data-flow model*. This is the case, for example, of the modeling of accelerator devices proposed in the WATERS 2019 challenge, where host-to-device and device-to-host data movements have to be exposed in explicit tasks, reducing the *programmability* of the system, as well as the *portability*, since the model is then tied to a specific architecture. Clearly, this strategy goes against MDE principles.

To avoid this inconvenience, and enhance the *programmability* and *portability* of the system, we take advantage of the *custom properties* available in AMALTHEA to describe two new characteristics in runnables:

- *Host parallelism:* This property describes a runnable as a potentially concurrent unit of work that is to be in the host system.
- *Accelerator parallelism:* This property describes a runnable as a potentially concurrent unit of work that is to be executed in and accelerator device (e.g., GPUs).

To illustrate the modifications implemented targeting performance, Figure 4a shows the workflow of a simple application enriched with information on how the application can be modeled with AMALTHEA tasks and runnables. The application runs $count$ times a pipeline composed of four steps: (1) read an image; (2) convert the image to a suitable format; (3) two different concurrent process on the image to produce results; and (4) merge and print the results. The figure also shows which parts are suitable for runnable granularity (green line for sequential runnables, and orange line for concurrent runnables, i.e., runnables not causing race conditions) or task granularity (yellow line), considering the AMALTHEA model.

Figure 4b shows the modeling of the application shown in Figure 4a using the extensions proposed for AMALTHEA in the form of custom properties. Specifically, the runnables corresponding to $AnalysisA$ and $AnalysisB$ are augmented with the *accelerator parallelism* property, while the rest of the runnables are augmented with the *host parallelism* property. Additionally, the model includes the data consumed and produced by each runnable in the form of labels. For example, runnable $AnalysisA$ consumes $image$ and produces $resultA$. This labels describe a data-flow execution model within the task that allows generating a Direct



(a) Application's workflow. (b) AMALTHEA model. (c) DAG (TDG).

Figure 4: Example of modeling parallelism with AMALTHEA.

Acyclic Graph (DAG) connecting the runnables and representing the parallelism exposed in the application. The DAG resulting from the model is depicted in Figure 4c.

This DAG (also called Task Dependency Graph, TDG, in OpenMP jargon) is the meta parallel programming abstraction that the code synthesis tool will generate for the compiler and analysis tools to generate and analyze code targeting performance, time predictability, energy efficiency and resilience. The interface used to define the TDG is described in *D6.2, Refined AMPERE ecosystem interfaces and integration plan* [4].

3 AMALTHEA code synthesis tool

The code synthesis tool used in AMPERE is the AMALTHEA synthetic load generator (SLG). It comes as a plugin of the APP4MC Eclipse project [8] used as domain specific modeling language (DSML). This section describes the augmentations added in the code synthesizer and the logic implemented for the automatic transformations from DSML to PPM to exploit performance. Further details on the extensions in the code synthesis tool are included in Section 9.1.

3.1 The AMALTHEA SLG

The AMALTHEA modeling software already includes a code synthesis tool capable of transforming the model into sequential C code. Runnables and tasks are created as C functions, while labels are transformed as global variables of the application. To simulate the OS scheduler the code generator makes use of Pthreads: one Pthread is created per each stimulus, and it runs indefinitely in an infinite loop with the stimuli periodicity, spawning the corresponding tasks.

As an illustration, Figure 5 shows snippets of the code generated for the application presented in Figure 4 by the AMALTHEA SLG, as provided by BOSCH.

3.2 Transformations targeting parallelism

We have modified the AMALTHEA SLG to include runnable functions into OpenMP `task` or `target` constructs, depending on the associated parallel property. Moreover, in order to properly define the `depend` synchronization clause and so avoid data races, we analyze the label usage (read or write) of all the runnables contained within an AMALTHEA task. This analysis allows to determine the parallelism exposed by the runnables and so determine the corresponding dependencies. It is important to remark that this analysis is based on the description of the usage of labels in the AMALTHEA model. Therefore, if runnables make an internal usage of labels different to the one describe in the model, race conditions may potentially occur[1]. The modifications implemented in the AMALTHEA SLG are described in Section 9.1.

Figure 6 exemplifies the transformations implemented in the augmented APP4MC SLG. Particularly, Figure 6a shows the model for code in Figure 4b highlighting the parts used by the AMALTHEA SLG to generate the code in Figure 6b. All runnables including the *host parallelism* custom property are annotated with an OpenMP `task` directive, i.e., run_read_image, $run\_convert\_image$ and $run\_merge\_results$. Similarly, those runnables including the *accelerator parallelism* custom property are annotated with an OpenMP `target` directive, i.e., $run\_analysisA$ and $run\_analysisB$. Furthermore, labels are transformed into dependency clauses, e.g., $run\_analysisA$ reads label $Image$ and writes label $ResultsA$, and the SLG lowers this information into a `depend(in:`*Image*`)` and a `depend(out:`*ResultsA*`)` clause, respectively.

[1] Chapter 5 presents the mechanisms needed to prevent race conditions, and so ensure a correct parallel execution.

```c
void *stepstimulusEntry(){
    Task1();
}

void *stepstimulusLoop(){
    pthread_t stepstimulus;
    for(;;){
        pthread_create(&
            stepstimulus, NULL
            ,
            stepstimulusEntry,
             NULL);
        usleep(1000000);
    }
}

int main(int argc, char **
     argv){
    pthread_t stepstimulus_;
    pthread_create(&
        stepstimulus_, NULL,
        stepstimulusLoop, NULL
        );
    pthread_join(stepstimulus_
        , NULL);
}
```

(a) Program entry point.

```c
void read_Image(int labelAccessStatistics) {
    int numberOfBytes = 62500;
    for (int repeat = 0 ; repeat <
         labelAccessStatistics; repeat++){
        int arraysize = sizeof(Image) / 4;
        int leftOverElements = arraysize % 10;
        int arraySizeWith10Multiples = arraysize -
             leftOverElements;
        int i = 0, a = 0;
        for (i = 0; i < arraySizeWith10Multiples;
             i = i + 10) {
            a = Image[i]; a = Image[i+1]; a =
                Image[i+2];
            a = Image[i+3]; a = Image[i+4]; a =
                Image[i+5];
            a = Image[i+6]; a = Image[i+7]; a =
                Image[i+8];
            a = Image[i+9];
        }
        for(; i<arraysize; i++) {
            a = Image[i];
        }
    }
}
```

(b) Read access to a label.

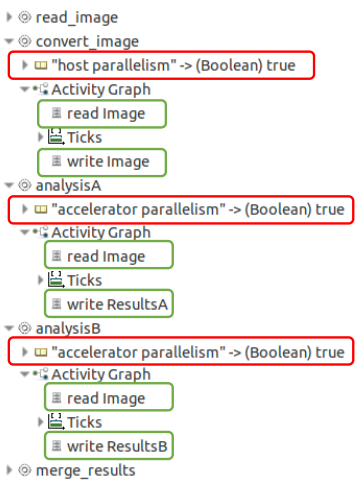
```c
void Task1(){
    run_read_image("");
    run_convert_image("");
    run_analysisA("");
    run_analysisB("");
    run_merge_results("");
}

void *Task1_entry(){ Task1();
    }
```

(c) A task.

```c
void run_analysisA(char* coreName){
    read_Image(1);
    executeTicks_DiscreteValueConstantImpl
        (1000000000);
    write_ResultsA(1);
}
```

(d) A runnable.

Figure 5: Automatically generated code with the *vanilla APP4MC SLG* for the AMALTHEA model in Figure 4b.



(a) AMALTHEA model augmented with parallelism features (as in Figure 4b).

```c
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out:Image)
  run_read_image("");

  #pragma omp task depend(inout:Image)
   depend(inout:Image)
  run_convert_image("");

  #pragma omp target depend(in:Image)
      depend(out:ResultsA)
  run_analysisA("");

  #pragma omp target depend(in:Image)
      depend(out:ResultsB)
  run_analysisB("");

  #pragma omp task depend(in:ResultsA,ResultsB)
  run_merge_results("");
}
```

(b) Automatically generated OpenMP code.

Figure 6: Sample transformation using the *augmented AMALTHEA SLG* for application in Figure 4a.

4 The meta parallel programming abstraction

The OpenMP code automatically generated by the code synthesis tool can be represented in the form of a task dependency graph (TDG). As an illustration, the TDG shown in Figure 4c holds the computation implemented in the code in Figure 6b. This representation is the meta parallel programming abstraction the AMPERE partners are using to communicate the different offline components of the software architecture, including the code synthesis tool, the compiler, and the different analysis tools.

This section firstly describes the generation of the TDG (implementation details on the component implemented during phase 2 of the project to extract the meta parallel programming model abstraction are provided in Section 9.2, and secondly summarizes provides a high-level description of the attributes envisioned for the analysis of the different non-functional requirements targeted in the project, including performance, time-criticality, energy efficiency and resilience. Furthermore, requirements to match the different DSMLs considered in the project are also described. The specific interface of the TDG is detailed in *D6.2, Refined AMPERE ecosystem interfaces and integration plan*.

4.1 Generation of the TDG

As described in deliverable D4.2 [9], the generation of the task dependency graph is implemented in the Mercurium [10] source-to-source compiler [11]. The overview of the implementation provided by Mercurium to support the generation of the TDG is shown in Figure 7. The compiler implements a frontend for C and C++ that generates a common internal representation (IR) for the two languages. This IR is later used in the middlend phases performing a series of analysis and optimizations. Then, the IR is lowered, generating C/C++ code augmented with calls to specific runtime libraries (e.g., GCC's libgomp in AMPERE's case). Finally, a native compiler is transparently called to produce the final binaries that will be executed in the system (e.g., GCC in AMPERE's case).

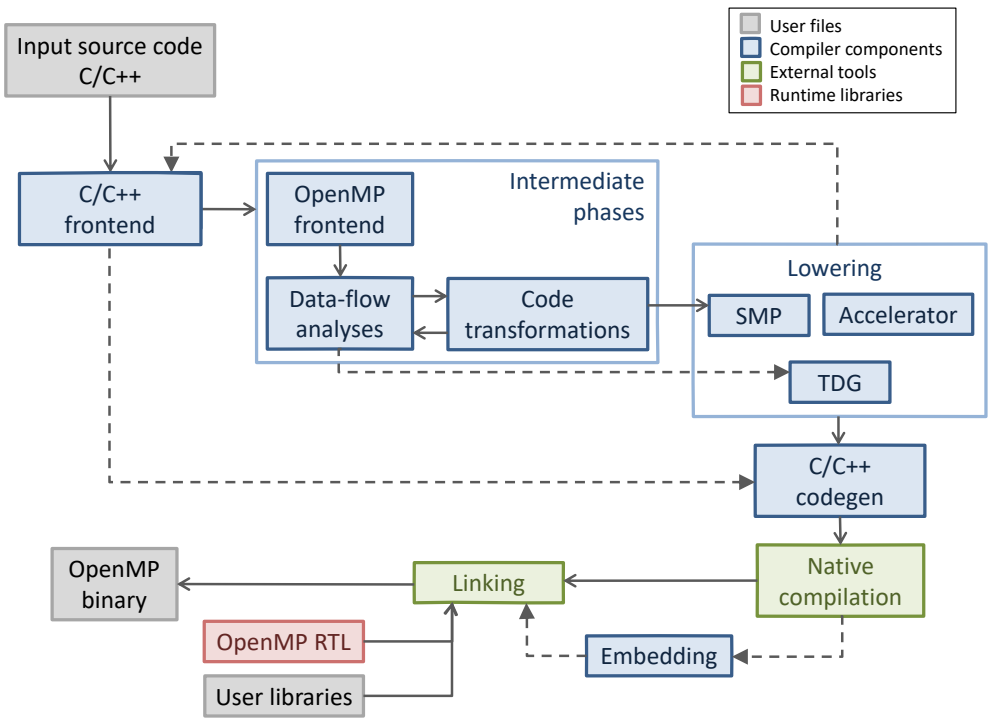


Figure 7: Mercurium pipeline for the generation of the TDG.

Figure 8 shows the TDG generated by Mercurium for the code in Figure 6b. This structure, further detailed

in deliverable 4.2, represents a sparse matrix containing all tasks in the system, and the dependencies among them. Furthermore, it contains definitions of the libgomp runtime functions (i.e., *gomp_set_tdg*) needed by the compiler for the final linking with the runtime library.

4.2 Augmentations for non-functional requirements

The original TDG generated by Mercurium contains the data structures needed by the GCC's libgomp runtime library to orchestrate a parallel execution driven by the TDG instead of being driven by the users code (i.e., loops, conditional statements and task dependencies). As a consequence, there is some information included in that TDG that is unnecessary from the AMPERE's offline analysis tools point of view for the analysis of the different non-functional requirements. The simplifications of Mercurium's TDG have been implemented, together with the augmentation of the TDG with information about performance. This work is done with a Python script further described in Section 9.2. Following subsections detail the modifications implemented for performance (the target at MS2), and envisioned for the rest of non-functional requirements, including time criticality, energy efficiency and resilience, as well as the information regarding DSML constraints that needs to be propagated to the offline analysis tools.

4.2.1 Performance

In order to analyze the performance of the system, we have enriched the basic TDG structure with information about the execution time of each OpenMP task, i.e., a node of the TDG. This information is extracted with the Extrae [12] tracing tool by executing the application while Extrae traces aspects like the instantiation and the execution of OpenMP tasks. Then, the Extrae traces are processed by a Python script (detailed in Section 9.2) that computes the total execution time of the tasks and enriches the TDG generated by the Mercurium source-to-source compiler with the execution times. Figure 9 shows the software components and the execution pipeline required for the generation of performance information. First the model is transformed into parallel code using OpenMP, as explained in Chapter 3. Then, Mercurium is used to generated two outputs: on one hand, the target code transforming OpenMP directives into calls to the target runtime library (i.e., libgomp); and on the other hand, the TDG that is later used by the parallel runtime library to orchestrate the parallel execution. Then, Extrae is used during the offline execution of the application to trace performance. Finally, a Python script joints the information of the original TDG with the information of the traces to generate the TDG to be used by the AMPERE's offline analysis tools.

Figure 10 shows an execution trace generated by Extrae for the code in Figure 6b after compiling with Mercurium+GCC and running with GCC's libgomp. By using the Python script described in Section 9.2, we are able to extract information about execution time and other performance counters, required by the different offline analysis tools of the AMPERE software ecosystem. Figure 11 shows the TDG resultant from processing the Mercurium's TDG shown in Figure 8, and the Extrae trace in Figure 10 with the proposed script. The execution time of each task is stored in the $execution\_time$ member of the $gomp\_tdg\_node$ structure, and the different papi counters gathered during the execution of the code are stored in the $papitot\_counter\_vals$, particularly: total number of instructions (42000050), total number of cycles (42000059), L1 cache misses (42000000), L2 cache misses (42000002), L3 cache misses (42000008), total branch instructions (42000055), conditional branches misspredictions (42000046), and total allocation stalled cycles (42001047).

In the future, we are considering to include in the TDG information about the amount of data produced and consumed by each task. This information can be extracted from the AMALTHEA model and from the automatically generated OpenMP code, as well. It will be useful to understand the cost of data movements an decide between host or accelerator implementations of a given runnable.

```
1  // File automatically generated
2  #include <stdio.h>
3  struct gomp_task;
4  struct gomp_tdg {
5     unsigned long id;          // Task instance ID
6     struct gomp_task *task_ptr;   // Pointer to the runtime task structure
7     unsigned short offin;      // Starting position within gomp_tdg_ins
8     unsigned short offout;     // Starting position within gomp_tdg_outs
9     unsigned char nin;         // Number of input dependencies
10    unsigned char nout;        // Number of output dependencies
11    signed char cnt;           // Number of dependent tasks:
12                               // (1) cnt == -1 Task executed or not created
13                               // (2) cnt == 0 task being executed
14                               // (3) cnt > 0 number of waiting tasks
15    int map;                   // Thread assigned to the task (static scheduler)
16    long task_counter;         // Private counter to compute execution time
17    long task_counter_end;     // Private counter to compute the final time
18    long runtime_counter;      // Private counter to compute runtime overhead
19    long taskpart_counter;     // Private counter to compute task part's time
20    struct gomp_tdg *next_waiting_tdg;   // Pointer for lazy task creation
21    unsigned int pragma_id;    // Identifier of the task contruct
22    void* data;                // Data required by the task
23 };
24
25 struct gomp_tdg gomp_tdg_0[5] = {
26    {.id = 1,.task_ptr = 0,.offin = 0,.offout = 0,.nin = 0,.nout = 1,.cnt = -1,.map = -1,.
          task_counter = 0,.task_counter_end = 0,.runtime_counter = 0,.taskpart_counter =
          0,.next_waiting_tdg = NULL,.pragma_id = 1},
27    {.id = 2,.task_ptr = 0,.offin = 0,.offout = 1,.nin = 1,.nout = 2,.cnt = -1,.map = -1,.
          task_counter = 0,.task_counter_end = 0,.runtime_counter = 0,.taskpart_counter =
          0,.next_waiting_tdg = NULL,.pragma_id = 2},
28    {.id = 3,.task_ptr = 0,.offin = 1,.offout = 3,.nin = 1,.nout = 1,.cnt = -1,.map = -1,.
          task_counter = 0,.task_counter_end = 0,.runtime_counter = 0,.taskpart_counter =
          0,.next_waiting_tdg = NULL,.pragma_id = 3},
29    {.id = 4,.task_ptr = 0,.offin = 2,.offout = 4,.nin = 1,.nout = 1,.cnt = -1,.map = -1,.
          task_counter = 0,.task_counter_end = 0,.runtime_counter = 0,.taskpart_counter =
          0,.next_waiting_tdg = NULL,.pragma_id = 4},
30    {.id = 5,.task_ptr = 0,.offin = 3,.offout = 5,.nin = 2,.nout = 0,.cnt = -1,.map = -1,.
          task_counter = 0,.task_counter_end = 0,.runtime_counter = 0,.taskpart_counter =
          0,.next_waiting_tdg = NULL,.pragma_id = 5}
31 };
32
33 unsigned short gomp_tdg_ins_0[] = {0, 1, 1, 3, 2};
34 unsigned short gomp_tdg_outs_0[] = {1, 3, 2, 4, 4};
35
36 // All TDGs are stored in a single data structure
37 unsigned gomp_num_tdgs = 1;
38 struct gomp_tdg *gomp_tdg[1] = {gomp_tdg_0};
39 unsigned short *gomp_tdg_ins[1] = {gomp_tdg_ins_0};
40 unsigned short *gomp_tdg_outs[1] = {gomp_tdg_outs_0};
41 unsigned gomp_tdg_ntasks[1] = {5};
42 unsigned gomp_maxI[1] = {0};
43 unsigned gomp_maxT[1] = {5};
44
45 // Initialize runtime data-strucures from here.
46 // This code is called from the compiler.
47 extern void GOMP_init_tdg(unsigned num_tdgs, struct gomp_tdg **tdg,
48                           unsigned short ** tdg_ins, unsigned short ** tdg_outs,
49                           unsigned *tdg_ntasks, unsigned *maxI, unsigned *maxT);
50 extern void GOMP_set_tdg_id(unsigned int);
51 void gomp_set_tdg(unsigned int tdg_id) {
52    GOMP_init_tdg(gomp_num_tdgs, gomp_tdg, gomp_tdg_ins, gomp_tdg_outs, gomp_tdg_ntasks,
          gomp_maxI, gomp_maxT);
53    GOMP_set_tdg_id(tdg_id);
54 }
```

Figure 8: TDG generated by Mercurium and consumed by the modified libgomp for the code in Figure 6b

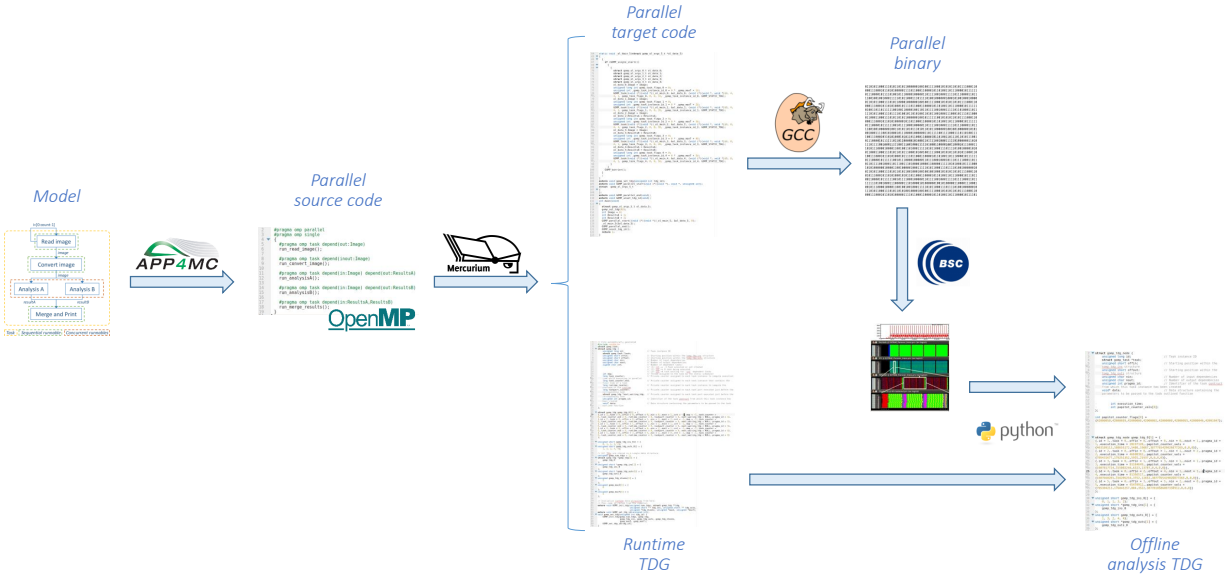


Figure 9: AMPERE's pipeline targeting performance optimizations.

Figure 10: Extrae trace

4.2.2 Time criticality

The information from the TDG required for timing analysis is related to performance traces and model references per node. The first requirement is the inclusion of information about the `Runnable` and the `Task` of the model the node of the graph relates to. This will be substantial for the analysis to directly map the performance traces to a `Runnable`.

Second, and most important for the analysis, is the actual performance traces of the OpenMP task. These traces should include the time elapsed of the task and a set of performance counters. The size of the set is specified per execution due to the fact that it will depend on the configuration defined for Extrae prior to the execution - or perf when dealing with runtime analysis - and on the actual performance counters the target platform has available. Linked to this set is also another collection that contains de identification of the respective performance counter.

Another attribute to consider in the TDG is the accessed labels of the model in each node. With the information of which and how many labels were accessed in a given `Runnable` it is possible to verify the performance impact of those labels (and the actual memory accesses) on the `Runnable`.

4.2.3 Energy efficiency

The TDG includes the necessary information for the energy analysis front-end described in D3.2, which produces the necessary information for the energy-efficiency optimization. First, the energy front-end uses the Extrae profiling information embedded in the TDG to extract the necessary characteristics of each individual OpenMP task. As the energy optimizer operates at an OpenMP task granularity, the structure of the task does in itself provide the necessary hooks to enable the profiler to run for all relevant components of the synthesized

```
struct gomp_tdg_node {
   unsigned long id;                    // Task instance ID
   unsigned short offin;                // Starting position within the tomp_tdg_ins
        structure
   unsigned short offout;               // Starting position within the tomp_tdg_outs
        structure
   unsigned char nin;                   // Number of input dependencies
   unsigned char nout;                  // Number of output dependencies
   unsigned int pragma_id;              // Identifier of the task contruct
   int execution_time;
   int papitot_counter_vals[8];
};

int papitot_counter_flags[8] =
     {42000050,42000059,42000000,42000002,42000008,42000055,42000046,42001047};

struct gomp_tdg_node gomp_tdg_0[5] = {
{.id = 1,.task = 0,.offin = 0,.offout = 0,.nin = 0,.nout = 1,.pragma_id = 1,.
     execution_time = 28197128,.papitot_counter_vals =
     {403189113,108655172,5488,19087,3877765429026877265,0,0,0}},
{.id = 2,.task = 0,.offin = 0,.offout = 1,.nin = 1,.nout = 2,.pragma_id = 2,.
     execution_time = 46698362,.papitot_counter_vals =
     {705431677,179251352,5925,21937,0,0,0,0}},
{.id = 3,.task = 0,.offin = 1,.offout = 3,.nin = 1,.nout = 1,.pragma_id = 3,.
     execution_time = 81164649,.papitot_counter_vals =
     {1007817724,315802244,6323,13787,0,0,0,0}},
{.id = 4,.task = 0,.offin = 2,.offout = 4,.nin = 1,.nout = 1,.pragma_id = 4,.
     execution_time = 81356517,.papitot_counter_vals =
     {1007660201,316205254,3752,12652,3877765429026877265,0,0,0}},
{.id = 5,.task = 0,.offin = 3,.offout = 5,.nin = 2,.nout = 0,.pragma_id = 5,.
     execution_time = 45478912,.papitot_counter_vals =
     {705304213,176841257,884,3522,3877810586007330912,0,0,0}}
};

unsigned short gomp_tdg_ins_0[] = {0, 1, 1, 3, 2};
unsigned short *gomp_tdg_ins[1] = {gomp_tdg_ins_0};
unsigned short gomp_tdg_outs_0[] = {1, 3, 2, 4, 4};
unsigned short *gomp_tdg_outs[1] = {gomp_tdg_outs_0};
```

Figure 11: TDG simplified and annotated with performance data

AMPERE system. Lastly, the energy optimizer makes use of the *EnergyBudget* custom attribute for tasks, to communicate the non-functional requirements with respect to the allowed energy envelope. This information is therefore forwarded from the AMALTHEA model to the energy front-end, by means of the TDG.

4.2.4 Resilience

Resiliency at the model level is being considered by augmenting selected AMALTHEA runnables with a *redundancy* custom property (as described in deliverables D3.2 [13], containing the description of the technique, and D4.2 [9], containing the implementation at runtime level). The redundant runnables are transformed into as many OpenMP tasks as specified in the model by the AMALTHEA SLG. Later, the compiler (particularly Mercurium) generated the TDG, which already contains these redundant tasks as extra nodes in the graph. By expressing this information in the structure of the TDG, it can be easily taken into account in performance, timing and energy analysis. However, if we see in the future that the redundant tasks have to be annotated in any manner, the TDG can easily be extended in the compiler.

4.2.5 DSML constraints

The DSMLs considered in the project (i.e., Amalthea and Capella) are being extended to support the requirements considered in the project. These requirements have to be included in the TDG for later consideration in

the offline analysis.

4.2.5.1 AMALTHEA

As presented previously in Figure 4, the runnables *analysisA* and *analysisB* were annotated with the property *accelerator parallelism*, meaning that these runnables are offloaded, in this case, for a GPU. This offloading can be observed in the code 6b, in lines 10 and 13, with the OpenMP keyword `target`.

The AMALTHEA extensions for offloading documented in the Deliverable 1.3 [3], complement the options for offloading described in this deliverable by including an FPGA as a target device. In the future, the current proposal presented in Figure 4 will be extended to capture in the model where the designer wants to offload the runnables, i.e., into a GPU or an FPGA. For instance, using the example in Figure 4, the runnable *analysisA* could be offloaded into a GPU while the runnable *analysisB* could be offloaded into an FPGA. This information shall be captured in the TDG, as well.

4.2.5.2 CAPELLA

In the Capella model, the information added in the context of the AMPERE project are the (Automotive) Safety Integrity Levels that can be assigned to software and hardware elements. This information will be modeled with an enumeration property value in Capella, and translated to a tag in Amalthea. It is an important information, since some safety requirements express constraints on the mapping between software and hardware elements based on the (A)SIL level of these elements (see Deliverable 1.2 [14]). These constraints shall be expressed in the Amalthea model, and then included in the TDG, so that they can be taken into account for schedulability analysis and resource allocation.

5 Assaying DSML correctness

Formal verification methods, including model checking and static analysis techniques, are typically used for the verification of safety critical embedded systems. This methods usually rely on complex verification oriented formal languages, and domain experts do not necessarily master these techniques. For this reason, verification tools tend to be embedded in the MDE framework, so they are transparent to the designer.

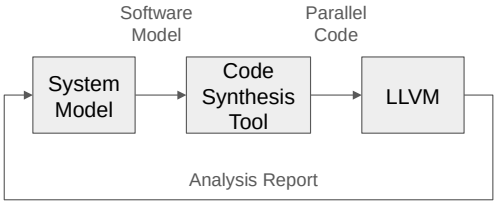
The introduction of parallel features in DSMLs requires support in the verification methods counterpart, in order to guarantee that the transformed parallel source code is free of race conditions. As exposed in Chapter 3, our transformation is guaranteed to be correct from the perspective of the usage of labels as defined in the AMALTHEA model. However, our analysis does not take into account how runnables actually access labels internally. This chapter addresses this topic.

In the HPC domain, several efforts have been made towards the analysis and enhancement of the safety of parallel systems. In the context of OpenMP, different works tackle the correctness of the task-based model. Of particular interest are the mechanisms for the automatic definition of the data-sharing and the dependency clauses of task directives [15, 16]. These techniques allow, on one hand, automatically defining these attributes and hence alleviating programmers from the burden of manually doing this job, and, on the other hand, checking the attributes defined by the programmer to verify the correctness of the program. Furthermore, techniques for the static analysis of OpenMP task-based programs considering inconsistencies in the data-sharing and dependency clauses, as well as the detection of race conditions [17, 18] also show good results enhancing the safety of parallel applications. Additionally, there are works that tackle the safety of the parallel framework with regard to the runtime implementation, and provide mechanisms for eliminating the use of dynamic memory [19], or reducing the overall memory consumption of the system [11]. These works aim at evolving the OpenMP runtime system to enable its qualification for the safety-critical domain. Finally, extensions to the OpenMP specification have also been proposed to enhance the analyzability of the parallel program [20], also facilitating its adoption in constrained environments.

Overall, the correctness techniques used for OpenMP in the context of HPC can be used for checking the model at the DSML level by virtue of the automatic transformation, from model to parallel programming model, implemented in the code synthesizer. This is, by validating the correctness of the task and target directives in terms of race conditions, and the correctness of the data-sharing, dependency and mapping clauses with regard to the access to variables within the tasks, we are able to validate the correctness of the custom properties for expressing parallelism (i.e., host parallelism and accelerator parallelism), and the labels expressed by the domain expert in each runnable at the AMALTHEA model level.

We have introduced two analyses implemented in the LLVM 11.1 compiler [21] (and further described in Section 9.3) to analyze the correctness of the data-sharing and the dependency clauses of OpenMP tasks, and thus detect possible errors in the AMALTHEA model. The compiler examines the source code inside OpenMP tasks, and detects the adequate data-sharing [15] and possible data races; next, this information used to determine the adequate dependency clauses [16], and the result is compared to the dependencies automatically inserted by the code synthesis tool to the OpenMP tasks in order to generate a final report. The report details if the possible data races are well covered by the current task's dependencies, as well as detecting useless dependencies. Since the code generator simply translates the label's usage into OpenMP task directives, any error detected with the correctness analyses comes from the system model.

The pipeline is represented in the Figure 12a, where a model is transformed to parallel source code and then analyzed by LLVM. Finally LLVM generates a report for all the runnables encapsulated into OpenMP tasks, being able to detect errors in the label's usage of the model. If this is the case, the report indicates the specific error. Figure 12b shows the report when the AMALTHEA model presented in Figure 4b does not define the label Image of runnable $convertimage$ as $write$.



(a) Pipeline.

```
1 Analyzing runnable: Image converting
2 Analyzing label: @Image
3     @Image detected scope: SHARED
4     Possible race condition, running Autodeps
5     ERROR: @Image should be OUT, check Software
          Model
```

(b) Sample report.

Figure 12: LLVM implementation for checking OpenMP tasks correctness with regard to data-races.

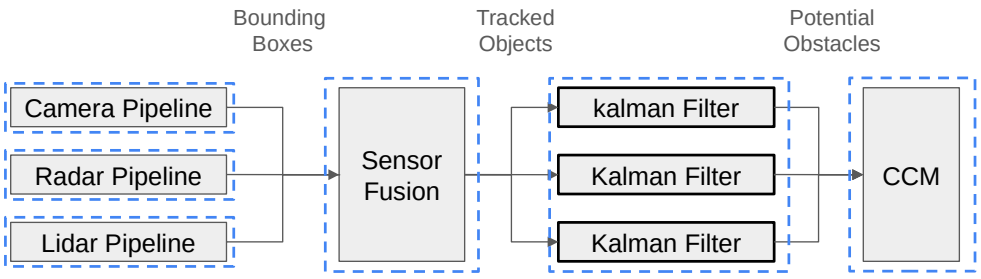
6 Evaluation

This section provides an evaluation of the benefits of combining parallel programming models with DSMLs, like AMALTHEA, in terms of programmability, portability and performance.

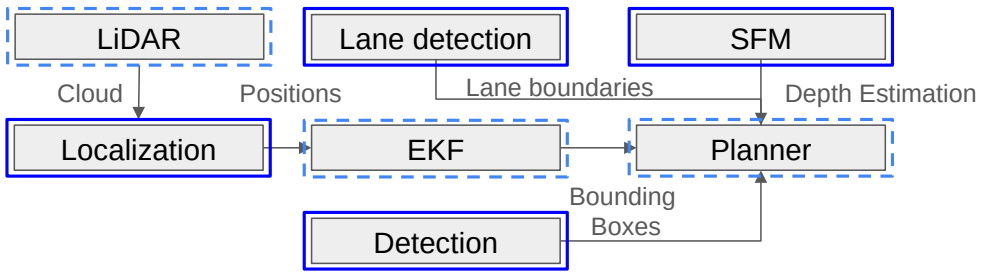
6.1 Experimental setup

Following we describe the settings for performing the evaluation:

- *Processor Architecure.* The experiments are performed on a NVIDIA Jetson TX2 board [22], which features a GPU, a 4-core ARM CPU and 8GB of main memory.
- *Applications.* We have selected two use cases, described with AMALTHEA, that are in charge of detecting obstacles through different sensors and track them to avoid possible collisions:
  - The Obstacle Detection and Avoidance System (ODAS) from the railway domain included in the AMPERE use-cases [14], and depicted in Figure 13a.
  - An autonomous driving application prototype from the WATERS 2019 challenge [23], depicted in Figure 13b.
- *Modeling and synthesis software.* To model the applications and to obtain the generated source code of the model we use the AMALTHEA 1.0 [6] platform. The code synthesis tool has been modified as specified in Chapter 3.
- *Compiler.* The generated OpenMP source code is analyzed and compiled using LLVM 11.1 [24].



(a) ODAS.



(b) WATERS2019.

Figure 13: LLVM implementation for checking OpenMP tasks correctness with regard to data-races.

6.2 Programmability and portability

Complex systems usually require to run kernels on accelerators to speed up the execution. The AMALTHEA model supports runnables assigned to a GPU by including extra tasks into the model that are in charge of

the data movements between the host and the accelerator, as proposed in the WATERS 2019 challenge, and depicted in Figure 14a [23].

The data movements are required by the accelerator to obtain the input data, and by the host to receive the final output. These are however implementation details related to the underlying processor architecture, exposed in the system model to allow considering the communications between the different devices in the timing analysis of the application. Moreover, exposing this aspects to the domain experts reduces the *programmability* of the system, as well as the *portability*, since the model is then tied to a specific architecture. Clearly, this strategy goes against MDE principles.

Instead, our approach allows hiding the communication between the host and the device by defining runnables with the*accelerator parallelism* property as shown in Figure 14b. Then, our code synthesis tool is responsible of transforming the accelerator runnable into an OpenMP target task, together with the data mapping clauses necessary to model the communication between the host and the accelerator. This information will hence be internally managed by the OpenMP runtime. Interestingly, previous works have already tackled the analysis of such a system implemented with OpenMP. More specifically, timing guarantees have been provided for heterogeneous systems composed of a multi-core host and a single accelerator device implemented with the OpenMP accelerator model [25].

(a) Original.

(b) Proposed.

Figure 14: AMALTHEA modeling of host-device communication.

This modification in the high-level description of the model has been applied in the autonomous driving use case. There, four different kernels are to run in a GPU: $localization$, $sensor\ fusion$, $object\ detection$, and $lane\ detection$. Each of these kernels, modeled as a unique runnable inside a task, originally has two extra dedicated tasks in the model managing the data transfers between the host and the GPU, one for sending data and another one for receiving data. In our model, these 8 dedicated tasks are removed, as well as the corresponding communications between the tasks, enhancing both the programmability and the portability of the model, while maintaining the potential performance.

6.3 Performance

The introduction of parallelism at the model level aims at enhancing the performance of the resulting system. This section evaluates the performance gain of the OpenMP source code resultant of the transformation of the two AMALTHEA models. On one hand, the ODAS application detects objects in an image by fusing the information coming from three sensors (camera, radar and lidar), and tracks them using a series of Kalman filters to compute their trajectories. Finally, a collision checker model determines whether a collision will potentially occur. We have modeled this system with AMALTHEA, as shown in Figure 13a, where the blue dotted boxes represent AMALTHEA tasks, and the internal grey boxes represent the runnables inside the tasks. Camera, radar, and lidar pipelines are inherently sequential, as well as the sensor fusion; each Kalman filter, instead, targets the trajectory of a detected object, and so all these runnables are concurrent. Kalman filters are fast

computation kernels that may benefit from the exploitation of fine-grain parallelism rather than the coarse-grain parallelism offered by AMALTHEA tasks. Therefore, we have augmented the ODAS original model with additional custom properties expressing the host parallelism inherent to the kalman filters. Consequently, Kalman filters can be executed in parallel with OpenMP because the code generator automatically translates those properties into OpenMP `task` directives and dependency clauses. The benefits obtained by introducing a simple custom property in each of the runnables representing Kalman filters are shown in Table 2. The baseline for computing the speedup is the (sequential) execution obtained with the model without the proposed custom properties. A 2.62x speedup is achieved when considering 4 threads and measuring only the portion of the parallel execution of kalman filters. This speed-up is reduced to 1.22x when taking into account the execution time of the whole application. When 2 threads are considered, the speedup achieved is 1.88x and 1.18x respectively. This modest performance speed up is due to the small contribution of the parallel execution of the kalman filters to the overall execution.

On the other hand, the WATERS challenge application, represented in Figure 13b, is composed of seven kernels, four of them executed in the GPU, i.e., $localization$, $sensor\ fusion\ (SFM)$, $object\ detection$, and $lane\ detection$. As described in Section 6.2, the original model has been modified by replacing the tasks dedicated to data movements between the host and the accelerator with runnables enclosed in the same task executing the computation in the accelerator (as described in Figure 14). The computation runnables have been further augmented with the *accelerator parallelism* custom property, and so the code synthesizer automatically transforms them into `target` directives with the proper `map` clauses. These target regions are then offloaded to the accelerator, and data copies are internally managed by the OpenMP framework (i.e., compiler and runtime systems). The benefits in performance obtained with the additional custom properties and augmented code synthesizer are shown in Table 2. The baseline for computing the speedup corresponds to the model without the proposed custom properties, and hence executed in the host. The results show a 6.21x speedup when considering only the parallelized kernels, and a 4.83x if we take into account the end-to-end execution time of the application.

Configuration		*Parallelized kernels*	*Full application*
ODAS	*2th*	1.88	1.18
	4th	2.62	1.22
WATERS	*4th & GPU*	6.21	4.83

Table 2: Performance evaluation in terms of speedup.

7 Conclusions

Our analyses, started in D2.1 [14] and further developed for this deliverable, conclude that state-of-the-art MDE standards, like AUTOSAR and AMALTHEA, are compatible with parallel programming models, like OpenMP, considering the execution model, the memory model, and the minimal support for non-functional requirements. Our proposed changes in the DSML, together with the augmented code synthesis tool that automatically transforms the enhanced model into parallel code, and a set of correctness analysis techniques applied at the level of the generated source code, allow maintaining the *correct-by-construction* paradigm of MDE technologies while exploiting parallel heterogeneous processor architectures.

Based on the analysis performed, this work provides an initial set of enhancements to exploit both host and accelerator parallelism using OpenMP. Our evaluation shows promising results in terms of programmability, portability, performance and correctness. As a result, we are working in further extensions, including the possibility of defining different implementations for a given runnable via a new *variant* property. This new feature, intended for enhancing the *portability* of the model, would allow to chose which version of the runnable to run based on the architecture underneath.

List of Acronyms

CPU	Central Processing Unit
D	Deliverable
DAG	Direct Acyclic Graph
DSML	Domain Specific Modeling Language
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
IR	Internal Representation
HPC	High-Performance Computing
MDE	Model-Driven Engineering
MS	Milestone
ODAS	Obstacle Detection Avoidance System
OS	Operating System
PPM	Parallel Programming Model
SLG	Synthetic Load Generator
T	Task
TDG	Task Dependency Graph
WP	Work Package

8 References

[1] European Comission and AMPERE beneficiaries, "Grant Agreement Description of Action," 2019.

[2] AMPERE, "D2.1. Model transformation requirements," 2020.

[3] ——, "D1.3. First release of the meta model-driven abstraction release," 2021.

[4] ——, "D6.2 Refined AMPERE ecosystem interfaces and integration plan," 2021.

[5] ——, "D1.1. System models requirements and use case selection," 2020.

[6] Eclipse Foundation, "Application Platform Project for MultiCore (APP4MC)," 2021, www.eclipse.org/app4mc/.

[7] OpenMP ARB, "The OpenMP API specification for parallel programming," 2021, www.openmp.org.

[8] Eclipse Foundation, "APP4MC SLG Linux," 2021, https://git.eclipse.org/c/app4mc/org.eclipse.app4mc.addon.transformation.git/.

[9] AMPERE, "D4.2. Independent run-time energy support, and predictability, segregation and resilience mechanisms," 2021.

[10] BSC, "Mercurium," 2021, https://pm.bsc.es/mcxx.

[11] R. E. Vargas, S. Royuela, M. A. Serrano, X. Martorell, and E. Quiñones, "A lightweight OpenMP4 run-time for embedded systems," in *Asia and South Pacific Design Automation Conference*. IEEE, 2016, pp. 43–49.

[12] BSC, "Extrae," 2021, https://tools.bsc.es/extrae.

[13] AMPERE, "D3.2. Single-criterion energy-optimization framework, predictable execution models, and software resilient techniques," 2020.

[14] ——, "D1.2. Analysis of functional safety aspects on single-criterion optimization and first release of the test bench," 2021.

[15] S. Royuela, A. Duran, C. Liao, and D. J. Quinlan, "Auto-scoping for OpenMP tasks," in *International Workshop on OpenMP*. Springer, 2012, pp. 29–43.

[16] S. Royuela, A. Duran, and X. Martorell, "Compiler automatic discovery of OmpSs task dependencies," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2012, pp. 234–248.

[17] J. Li, D. Hei, and L. Yan, "Correctness analysis based on testing and checking for OpenMP programs," in *ChinaGrid Annual Conference*. IEEE, 2009, pp. 210–215.

[18] S. Royuela, R. Ferrer, D. Caballero, and X. Martorell, "Compiler analysis for OpenMP tasks correctness," in *ACM International Conference on Computing Frontiers*, 2015, pp. 1–8.

[19] A. Munera, S. Royuela, and E. Quiñones, "Towards a qualifiable OpenMP framework for embedded systems," in *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2020, pp. 903–908.

[20] S. Royuela, A. Duran, M. A. Serrano, E. Quiñones, and X. Martorell, "A Functional Safety OpenMP* for Critical Real-Time Embedded Systems," in *International Workshop on OpenMP*. Springer, 2017, pp. 231–245.

[21] A. Munera, S. Royuela, R. Ferrer, R. Peñacoba, and E. Quiñones, "Static Analysis to Enhance Programmability and Performance in OmpSs-2," in *International Conference on High Performance Computing*. Springer, 2020, pp. 19–33.

[22] NVIDIA, "Jetson TX2," 2019, https://developer.nvidia.com/embedded/jetson-tx2.

[23] Hamann, Arne and Dasari, Dakshina and Wurst, Falk and Saudo, Ignacio and Capodieci, Nicola and Burgio, Paolo and Bertogna, Marko., "WATERS Industrial Challenge," 2019, https://www.ecrts.org/forum/download/file.php?id=100&sid=93986ba3a054d452b1222b05545facc3.

[24] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[25] M. A. Serrano and E. Quiñones, “Response-time analysis of dag tasks supporting heterogeneous computing,” in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

9 Appendix

This section includes the implementation details of the software developed for fulfilling MS2 in WP2.

9.1 AMALTHEA SLG

Figure 17 shows the extensions implemented in the Amalthea SLG for generating OpenMP code targeting performance and resiliency, based on an Amalthea model augmented with the extensions presented in Section 2, for performance, and also those presented in D3.2 for resiliency.

```
1  private boolean processRunnable(
2      Runnable runnable, List<String> statements,
3      List<Label> externLabels, StringBuilder puDefinition) {
4
5      boolean hasParallelRunnables = false;
6      int duplicates = 0;
7
8      if (runnable.getCustomProperties().containsKey("redundancy")) {
9          Value V = runnable.getCustomProperties().get("redundancy");
10         EStructuralFeature feature = V.eClass().getEStructuralFeatures().get(0);
11         duplicates = (int) V.eGet(feature);
12     }
13
14     if (runnable.getCustomProperties().containsKey("accelerator parallelism")) {
15         Value V = runnable.getCustomProperties().get("accelerator parallelism");
16         EStructuralFeature feature = V.eClass().getEStructuralFeatures().get(0);
17         boolean acc_par = (boolean) V.eGet(feature);
18         if(acc_par) {
19             ArrayList<Label> LabelsOut = new ArrayList<Label>();
20             ArrayList<Label> LabelsIn = new ArrayList<Label>();
21             ArrayList<Label> LabelsInOut = new ArrayList<Label>();
22             for (ActivityGraphItem runitem : runnable.getActivityGraph().getItems()) {
23                 if(runitem instanceof LabelAccess) {
24                     LabelAccess la = (LabelAccess) runitem;
25                     Label label = la.getData();
26                     if(!externLabels.contains(label)) {
27                         externLabels.add(label);
28                     }
29                     if(la.getAccess() == LabelAccessEnum.READ) {
30                         if (!LabelsOut.contains(label) && !LabelsIn.contains(label))
31                             LabelsIn.add(label);
32                     }
33                     if(la.getAccess() == LabelAccessEnum.WRITE) {
34                         if(!LabelsOut.contains(label))
35                             LabelsOut.add(label);
36                     }
37                 }
38             }
39
40             ArrayList<Label> ToRemove = new ArrayList<Label>();
41             for (Label inLabel: LabelsIn) {
42                 if(LabelsOut.contains(inLabel)) {
43                     ToRemove.add(inLabel);
44                     LabelsInOut.add(inLabel);
45                 }
46             }
47             LabelsIn.removeAll(ToRemove);
48             LabelsOut.removeAll(ToRemove);
49             String isOpenMP= "\n#if defined(_OPENMP)";
50             String isOMPSS= "\n#elif defined(_OMPSS)";
51             String OpenMPPragma = "\n #pragma omp target";
52             String OmpssPragma= "\n #pragma oss target";
```

```
            if(LabelsIn.size()!=0) {
                String depends = "", maps="";
                depends+="␣depend(in:";
                maps+="␣map(to:";
                for (Label inLabel: LabelsIn) {
                    depends+=inLabel.getName() +",";
                    maps+=inLabel.getName() + "[0:"+ AmaltheaModelUtils.getLabelArraySize(
                        inLabel)+"],";
                }
                depends= depends.substring(0, depends.length()-1);
                maps= maps.substring(0, maps.length()-1);
                depends+=")";
                maps+=")";
                OpenMPPragma= OpenMPPragma + depends;
                OmpssPragma= OmpssPragma + depends + maps;
            }
            if(LabelsOut.size()!=0) {
                String depends = "", maps="";
                depends+="␣depend(out:";
                maps+="␣map(from:";
                for (Label outLabel: LabelsOut) {
                    depends+=outLabel.getName() +",";
                    maps+=outLabel.getName() + "[0:"+ AmaltheaModelUtils.getLabelArraySize
                        (outLabel)+"],";
                }
                depends= depends.substring(0, depends.length()-1);
                maps= maps.substring(0, maps.length()-1);
                depends+=")";
                maps+=")";
                OpenMPPragma= OpenMPPragma + depends;
                OmpssPragma= OmpssPragma + depends + maps;
            }
            if(LabelsInOut.size()!=0) {
                String depends = "", maps="";
                depends+="␣depend(inout:";
                maps+="␣map(tofrom:";
                for (Label inoutLabel: LabelsInOut) {
                    depends+=inoutLabel.getName() +",";
                    maps+=inoutLabel.getName() + "[0:"+ AmaltheaModelUtils.
                        getLabelArraySize(inoutLabel)+"],";
                }
                depends= depends.substring(0, depends.length()-1);
                maps= maps.substring(0, maps.length()-1);
                depends+=")";
                maps+=")";
                OpenMPPragma= OpenMPPragma + depends;
                OmpssPragma= OmpssPragma + depends + maps;
            }
            OmpssPragma +="␣copy_deps";
            statements.add(isOpenMP + OpenMPPragma + isOMPSS + OmpssPragma + "\n#endif");
        }
    }
    if (runnable.getCustomProperties().containsKey("host␣parallelism")) {
        Value V = runnable.getCustomProperties().get("host␣parallelism");
        EStructuralFeature feature = V.eClass().getEStructuralFeatures().get(0);
        Boolean host_par = (Boolean) V.eGet(feature);
        if(host_par) {
            ArrayList<Label> LabelsOut = new ArrayList<Label>();
            ArrayList<Label> LabelsIn = new ArrayList<Label>();
            ArrayList<Label> LabelsInOut = new ArrayList<Label>();
            for (ActivityGraphItem runitem : runnable.getActivityGraph().getItems()) {
                if(runitem instanceof LabelAccess) {
                    LabelAccess la = (LabelAccess) runitem;
                    Label label = la.getData();
                    if(!externLabels.contains(label))
                        externLabels.add(label);
                    if(la.getAccess() == LabelAccessEnum.READ) {
                        if (!LabelsOut.contains(label) && !LabelsIn.contains(label))
                            LabelsIn.add(label);
                    }
```

```
                    if(la.getAccess() == LabelAccessEnum.WRITE) {
                        if(!LabelsOut.contains(label))
                        LabelsOut.add(label);
                    }
                }
            }
            ArrayList<Label> ToRemove = new ArrayList<Label>();
            for (Label inLabel: LabelsIn) {
                if(LabelsOut.contains(inLabel)) {
                    ToRemove.add(inLabel);
                    LabelsInOut.add(inLabel);
                }
            }
            LabelsIn.removeAll(ToRemove);
            LabelsOut.removeAll(ToRemove);
            String isOpenMP= "\n#if␣defined(_OPENMP)";
            String isOMPSS= "\n#elif␣defined(_OMPSS)";
            String OpenMPPragma = "\n␣#pragma␣omp␣task";
            String OmpssPragma= "\n␣#pragma␣oss␣task";
            String depends="";
            if(LabelsIn.size()!=0) {
                depends+="␣depend(in:";
                for (Label inLabel: LabelsIn) {
                    depends+=inLabel.getName() +",";
                }
                depends= depends.substring(0, depends.length()-1);
                depends+=")";
            }
            if(LabelsOut.size()!=0) {
                depends+="␣depend(out:";
                for (Label outLabel: LabelsOut) {
                    depends+=outLabel.getName() + ",";
                }
                depends= depends.substring(0, depends.length()-1);
                depends+=")";
            }
            if(LabelsInOut.size()!=0) {
                depends+="␣depend(inout:";
                for (Label inoutLabel: LabelsInOut) {
                        depends+=inoutLabel.getName() + ",";
                }
                depends= depends.substring(0, depends.length()-1);
                depends+=")";
            }
            if(duplicates>0)
                depends+=("␣redundancy("+duplicates+")");
            OpenMPPragma+= depends;
            OmpssPragma+=depends;
            statements.add(isOpenMP + OpenMPPragma + isOMPSS + OmpssPragma + "\n#endif");
            hasParallelRunnables = true;
        }
    }
    statements.add("run_" + runnable.getName() + "(\"" + puDefinition.toString() + "\");")
        ;
    return hasParallelRunnables;
}
```

Figure 17: Modifications for supporting parallelism in the APP4MC SLG.

9.2 Python script for Extrae/TDG performance analysis

Figure 21 shows script that parses a Extrae trace and extracts information about the performance of each OpenMP tasks to later add it to the TDG generated by the Mercurium compiler.

```
import re
import collections
import sys
import copy

if(len(sys.argv)) < 4: #usage: ./script *.prv *_tdg.c *.pcf
    quit()
prvFile = sys.argv[1] #*.prv: execution data measured by Extrae
tdgFile = sys.argv[2] #*_tdg.c TDG structure generated by Mercurium
pcfFile = sys.argv[3] #*.pcf: list of Extrae configuration flags
if(not(prvFile.endswith(".prv")) or not(tdgFile.endswith("tdg.c")) or not(pcfFile.
    endswith(".pcf"))):
    print("\n Wrong input files. Expecting, in order: *.prv *_tdg.c *.pcf. Exiting. \n")
    quit()

#Parse the *.prv file (The prv format is described in D6.2)
#we are interested in events between task execution begin and task execution end
lines = []
with open(prvFile) as prvInput:
    lines = prvInput.readlines()
papiEventLines = ""
with open(pcfFile) as pcfInput:
    papiEventLines = pcfInput.read()

#the flag for the hardware event counters starts always with 42
papiEventsFlagList = re.findall(r'42[0-9]{6}', papiEventLines)
#exampe for papi event flags, from the pcf
#7  42000050 PAPI_TOT_INS [Instr completed]
#7  42000059 PAPI_TOT_CYC [Total cycles]
#7  42000000 PAPI_L1_DCM [L1D cache misses]
#7  42000002 PAPI_L2_DCM [L2D cache misses]
#7  42000008 PAPI_L3_TCM [L3 cache misses]
#papiEventFlags = ['42000050', '42000059', '42000000', '42000002', '42000008']
papiEventFlags = papiEventsFlagList
capturingEventTimeFlag = '60000023' #task execution flag
capturingEventIdFlag = '60000028' #task id flag
captureEventFlags = [capturingEventIdFlag, capturingEventTimeFlag] + papiEventFlags
captureEventValues = {} #values for the flags in captureEventFlags, but mapped per thread
numOfPapiEvents = len(papiEventFlags)
numOfValuesToCapture = len(captureEventFlags)  #event id, event time, all papi event
    counters
captureEventIdIndx = 0 #for the ease of reading
captureEventTimeIndx = 1
allCapturesCollection = [] #the final result of parsing extrae outputs

def initCountersIfNeeded(captureEventValues, thrId):
    if(thrId not in captureEventValues):
        captureEventValues[thrId] = [0] * numOfValuesToCapture

def resetCounters(captureEventValues, thrId):
    if(thrId in captureEventValues):
        captureEventValues[thrId] = [0] * numOfValuesToCapture
    else:
        print("Error! Counters should already be initialized for this thread, before reset.
            ")
        quit()
```

```
#PAPI counters show the sum of all event counts since the last event execution
def incrementCounterValuesForEventInThread(captureEventValues, thrId, entry):
    papiEventIndx = 0
    while papiEventIndx < len(papiEventFlags):
        papiEvent = papiEventFlags[papiEventIndx]
        #papiEventIndx is the same index for the list of flags as for the list of values
        indx = 0
        while(indx < len(entry)):
            #the value of the papi counter is located right after the papi event id in the prv entry:
            if(papiEvent == entry[indx]):
                #the rest have to be added up because they measure events since the last measurement
                captureEventValues[thrId][2 + papiEventIndx] = captureEventValues[thrId][2 + papiEventIndx] + int(entry[indx+1])
                break; #catch only in first papi event val, that is in Next Evt Val
            indx += 1
        papiEventIndx += 1

def isCaptureEventStart(entry):
    if((capturingEventIdFlag in entry) and (capturingEventTimeFlag in entry)):
        return True
    else:
        return False

def isCaptureEventEnd(entry):
    if( (capturingEventTimeFlag in entry) and not (capturingEventIdFlag in entry) and (entry[entry.index(capturingEventTimeFlag)+1] == '0') ):
        return True
    else:
        return False

#capture start event (60000023)
def startEventCapture(captureEventValues, thrId, entry):
    #the captureEventValues is a running counter of all values
    #but shouldn't capture events when not inside the execution of a task
    if((thrId in captureEventValues) and (captureEventValues[thrId][0] != 0) and (captureEventValues[thrId][1] != 0)):
        print("Ending of previous capture event has not reset the counters. Aborting...")
        quit()
    resetCounters(captureEventValues, thrId)
    captureEventValues[thrId][captureEventIdIndx] = entry[entry.index(capturingEventIdFlag) + 1]
    captureEventValues[thrId][captureEventTimeIndx] = int(entry[entry.index(capturingEventTimeFlag) - 1])

def endEventCapture(captureEventValues, thrId, entry):
    #do nothing for captureEventValues[thrId][0]
    captureEventValues[thrId][captureEventTimeIndx] = int(entry[entry.index(capturingEventTimeFlag) - 1]) - captureEventValues[thrId][captureEventTimeIndx]  #calculate execution time
    incrementCounterValuesForEventInThread(captureEventValues, thrId, entry)
    allCapturesCollection.append(captureEventValues[thrId])
    #finished mapping counters on this thread in this timeframe to this capture event
    resetCounters(captureEventValues, thrId)

#add up all PAPI counters during the execution time of an event
def processOtherEvents(captureEventValues, thrId, entry):
    incrementCounterValuesForEventInThread(captureEventValues, thrId, entry)

entry = []
for l in lines:
    entry = l.strip().split(':')
    thrId = int(entry[4])
    initCountersIfNeeded(captureEventValues, thrId)
    if(isCaptureEventStart(entry)):
        startEventCapture(captureEventValues, thrId, entry)
    elif(isCaptureEventEnd(entry)):
        endEventCapture(captureEventValues, thrId, entry)
    else:
        processOtherEvents(captureEventValues, thrId, entry)
```

```
#parse the tdg.c file, to search for data to modify with the execution times
tdgLines = ''
tdgStruct =''
with open(tdgFile) as tdgInput:
    tdgLines = tdgInput.read()
nameOfExecutionTime = "execution_time"
#PAPI flags and values list will have identically ordered elements
nameOfPapiFlags = "papitot_counter_flags"
nameOfPapiVals = "papitot_counter_vals"
#small cleanup of the tdg values (needed for execution) that we don't need:
def isMemberToRemove(member):
    if(("task_counter" in member) or ("task_counter_end" in member) or ("runtime_counter"
        in member) or ("taskpart_counter" in member) or ("map" in member) or ("cnt" in
        member) or ("next_waiting_tdg" in member)):
        return True
    else:
        return False

#parse the tdg structure and get the content of the body of the tdg struct declaration
matchTdgStructDecl = re.search(r'struct[\s]*gomp_tdg[\s]*\{([\s\S]*?)\}[\s]*;', tdgLines)
tdgStructDeclBody = matchTdgStructDecl.group(1)
tdgStructDeclBodyNew = copy.copy(tdgStructDeclBody) + "\n\n\t␣int␣" + nameOfExecutionTime
    + ";␣\n\t␣int␣" + nameOfPapiVals + "[" + str(numOfPapiEvents) + "];␣\n\n"

#assuming that member declarations are in separate lines
tdgStructDeclBodyNewLines = tdgStructDeclBodyNew.splitlines()
for member in tdgStructDeclBodyNew.splitlines():
    if(isMemberToRemove(member)):
        tdgStructDeclBodyNewLines.remove(member)

tdgStructDeclBodyNew = '\n'.join(tdgStructDeclBodyNewLines)
tdgStructDeclWhole = matchTdgStructDecl.group(0)

tdgStructDeclWholeNew = tdgStructDeclWhole.replace(tdgStructDeclBody,
    tdgStructDeclBodyNew)
tdgStructDeclWholeNewPlusPapiFlagDecl = tdgStructDeclWholeNew + "\n\n" + "int␣" +
    nameOfPapiFlags + "[" + str(numOfPapiEvents) + "]␣=␣{" + ','.join(papiEventFlags) + "
    };␣\n\n"

tdgOutputLines = "\n" + tdgStructDeclWholeNewPlusPapiFlagDecl + "\n"

listAllTdgStructs = re.findall( \
    r'struct[\s]*gomp_tdg[\s]*gomp_tdg_[0-9]+\[.*?\][\s]*=[\s]*\{[\s\S]*?\}[\s]*;',
        tdgLines)
numOfTdgs = len(listAllTdgStructs)
for tdgStructInList in listAllTdgStructs:
    #get the content of the body of the tdg struct definition
    matchTdgStruct = re.search(\
        r'struct[\s]*gomp_tdg[\s]*gomp_tdg_[0-9]+\[.*?\][\s]*=[\s]*\{([\s\S]*?)\}[\s]*;',
            tdgStructInList)
    tdgStruct = matchTdgStruct.group(1)
    #parse it into dictionary with keys as task ids and values as the list of vars
        describing that node
    oldTdg = {}
    tdgNode = []
    tdg = {}

    for matchTdgNode in re.finditer(r'\{(.*?)\}', tdgStruct):
        tdgNode = matchTdgNode.group(1).split(',')
        matchId = re.search(r'\.id.*?(\d+)', tdgNode[0]) #id is the first member of the tdg
            node struct
        tdgId = int(matchId.group(1))
        oldTdg[tdgId] = tdgNode
        #remove: task, task_counter, task_counter_end, runtime_counter, taskpart_counter,
            cnt, map
        tdg[tdgId] = copy.copy(tdgNode)
        for member in tdgNode:
            if(isMemberToRemove(member)):
                tdg[tdgId].remove(member)
```

```
        for capture in allCapturesCollection:
            if(capture[captureEventIdIndx].strip() == str(tdgId)):
                tdg[tdgId].append("." + nameOfExecutionTime + " = " + str(capture[
                    captureEventTimeIndx]))
                tdg[tdgId].append("." + nameOfPapiVals + " = {" + ','.join(list(map(str,
                    capture[2:]))) + "}")

    #bring it all together into output tdg.c file
    tdgForWrite = []
    tdgCombinedLines = tdgStructInList
    for node in tdg:
        if node not in oldTdg:
            print("Old and new TDGs must have same nodes!")
            quit()
        tdgCombinedLines = tdgCombinedLines.replace("{" + ','.join(oldTdg[node]) + "}", "{"
            + ','.join(tdg[node]) + "}")

    #parses dependencies:
    matchTdgStructDependIns = re.findall(r'unsigned.*?gomp_tdg_ins[\s\S]*?;', tdgLines)
    matchTdgStructDependOuts = re.findall(r'unsigned.*?gomp_tdg_outs[\s\S]*?;', tdgLines)
    tdgLinesDependInsAndOuts = '\n' + '\n'.join(matchTdgStructDependIns) + '\n' + '\n'.
        join(matchTdgStructDependOuts) + '\n'

    tdgOutputLines = tdgOutputLines + "\n\n" + tdgCombinedLines
    tdgOutputLines = re.sub(r'struct[\s]*gomp_tdg\s', 'struct gomp_tdg_node ',
        tdgOutputLines)
    tdgOutputLines = re.sub(r'struct[\s]*gomp_tdg\{', 'struct gomp_tdg_node{',
        tdgOutputLines)
    tdgOutputLines = re.sub(r'struct[\s]*gomp_tdg;', 'struct gomp_tdg_node;',
        tdgOutputLines)
    tdgOutputLines = tdgOutputLines + '\n' + tdgLinesDependInsAndOuts

indx = tdgFile.index("tdg.c")
tdgOutputFile = tdgFile[:indx] + "simple_output_" + tdgFile[indx:]
tdgOutput = open(tdgOutputFile, "w+t")
tdgOutput.write(tdgOutputLines)
tdgOutput.close()
```

Figure 21: Python script for performance processing of Extrae traces and TDG augmentation.

9.3 LLVM compiler for OpenMP correctness

Figure 33 shows the analysis implemented in the LLVM compiler targeting correctness.

```
for (auto &SelectedTask : FI.TaskFuncInfo.PostOrder) {
  bool AutoDeps = true;

  setPreSync(SelectedTask, SyncPoints, OI, FI.TaskFuncInfo.PostOrder);
  setPostSync(SelectedTask, SyncPoints, OI, FI.TaskFuncInfo.PostOrder);
  setConcurrentTasks(SelectedTask, OI, FI.TaskFuncInfo);
  setConcurrentSequential(SelectedTask, OI, FI.TaskFuncInfo, SyncPoints, FI.TaskFuncInfo
      .PostOrder);

  // Print concurrent code blocks
  printTaskConcurrentBlocks(SelectedTask);

  SmallVector<std::pair<Value *, DSAValue>, 10> ScopedVariables;

  // Clean scope of variables
  for (Value *Var : SelectedTask.DSAInfo.Firstprivate)
    ScopedVariables.push_back(std::pair<Value *, DSAValue>(Var, FIRSTPRIVATE));
  SelectedTask.DSAInfo.Firstprivate.clear();

  for (Value *Var : SelectedTask.DSAInfo.Private)
    ScopedVariables.push_back(std::pair<Value *, DSAValue>(Var, PRIVATE));
  SelectedTask.DSAInfo.Private.clear();

  for (Value *Var : SelectedTask.DSAInfo.Shared)
    ScopedVariables.push_back(std::pair<Value *, DSAValue>(Var, SHARED));
  SelectedTask.DSAInfo.Shared.clear();

  //Requires so global variables used in the task are added as variables to analyze,
      even they are not detected by Clang
  for (auto &Global : F.getParent()->getGlobalList()){
     Value *Variable = dyn_cast<Value>(&Global);

     for (User *U : Variable->users()){
        SmallVector<Instruction *, 4> CallList;
        SmallPtrSet<Function *, 10> AnalyzedFunctions;

        if(Instruction *I = dyn_cast<Instruction>(U)){
           obtainCallsInside(I, SelectedTask.Entry, SelectedTask.Exit ,OI,
               AnalyzedFunctions, CallList);
        }
        else{
           for (User *UNew : U->users()) {
              if(Instruction *I = dyn_cast<Instruction>(UNew)){
                 obtainCallsInside(I, SelectedTask.Entry, SelectedTask.Exit, OI,
                     AnalyzedFunctions, CallList);
              }
           }
        }
        //If there is a call, that means that the variable is used inside the task
        if(CallList.size()){
           bool exists= false;
           for (auto AlreadyVariable : ScopedVariables){
              if(AlreadyVariable.first==Variable){
                 exists=true;
                 break;
              }
           }
           if(!exists)
              ScopedVariables.push_back(std::pair<Value *, DSAValue>(Variable, SHARED));
           break;
        }
     }
  }
```

```
// Count scope of variables
int numUnknown = 0, numShared = 0, numFirstPrivate = 0, numPrivate = 0;
for (auto Variable : ScopedVariables) {
   Value *Var = Variable.first;
   dbgs() << "\n\033[1mAnalyzing␣variable:␣\033[0m";
   Var->printAsOperand(dbgs(), false);
   dbgs() << "\n";

   // Declare important attributes for the task, to be filled and used in the
       algorithm
   bool AutoDepsActivated = false;
   DSAValue OriginalDSA = Variable.second;
   DSAValue CorrectedDSA = UNDEF;
   VarUse UsedInConcurrent = NOT_USED;
   VarUse UsedInTask = NOT_USED;
   bool IsGlobalVariable = false;
   bool IsComposited;

   // Find if the variable is composited
   if (Var->getType()->isPointerTy())
      IsComposited = dyn_cast<CompositeType>(Var->getType()->getPointerElementType());
   else
      IsComposited = false;

   // Find if the variable is a Pointer
   bool IsPointer;
   if (Var->getType()->isPointerTy())
      IsPointer = Var->getType()->getContainedType(0)->isPointerTy();

   // Vector for storing variable uses in concurrent blocks, inside and outside tasks
   SmallVector<ValueAccess, 4> UsedInTaskValues;
   SmallVector<ValueAccess, 4> UsedInConcurrentValues;

   // Dummy ValueAccess, temporal fix
   ValueAccess dummy;

   for (User *U : Var->users()) {
      if (Instruction *I = dyn_cast<Instruction>(U)) {
         SmallPtrSet<Function *, 10> AnalyzedFunctions;
         int TaskContainsCall=UseIsInExternalCall(I, SelectedTask.Entry, SelectedTask.
             Exit, OI, AnalyzedFunctions);

         // Use is inside the task
         if ((OI.dominates(SelectedTask.Entry, I) && !OI.dominates(SelectedTask.Exit,
             I)) || TaskContainsCall) {
            AnalyzedFunctions.clear();
            valueInInstruction(I, Var, UsedInTask, false, &AnalyzedFunctions,
                UsedInTaskValues, true, dummy, BAA, NONE, true);
         }
         for (ConcurrentBlock &Block : SelectedTask.ConcurrentBlocks) {
            AnalyzedFunctions.clear();
            bool TaskContainsCall=UseIsInExternalCall(I, Block.Entry, Block.Exit, OI,
                AnalyzedFunctions);

            // Use is inside a concurrent block
            if ((OI.dominates(Block.Entry, I) && !OI.dominates(Block.Exit, I)) ||
            I == Block.Entry || I == Block.Exit || TaskContainsCall) {
               AnalyzedFunctions.clear();
               int previousSize = UsedInConcurrentValues.size();
               VarUse PreviousUsedInConcurrent = UsedInConcurrent;
               valueInInstruction(I, Var, UsedInConcurrent, false, &AnalyzedFunctions,
                    UsedInConcurrentValues, true, dummy, BAA, NONE, true);
               int numAdditions= UsedInConcurrentValues.size() - previousSize;
               // Check if the use was inside a task or not
               if(Block.Entry->getFunction() == I->getFunction()){
                  for (auto ThisTask : FI.TaskFuncInfo.PostOrder) {
                     if ((OI.dominates(ThisTask.Entry, I) && !OI.dominates(ThisTask.
                         Exit, I))) {
                        //Check if exists a taskdependency, if it exists we remove the
                            use since is no concurrent anymore
                        if(isPotentiallyReachable(SelectedTask.Exit, ThisTask.Entry)
                            && !existsTaskDependency(ThisTask, SelectedTask)){
                           UsedInConcurrentValues.back().ConcurrentUseInTask = true;
                           break;
                        }
```

```
                        else{
                            UsedInConcurrentValues.pop_back();
                            break;
                        }
                    }
                }
            }
            else{
                //Obtain a vector of all calls inside the concurrent block
                SmallVector<Instruction *, 4> CallList;
                AnalyzedFunctions.clear();
                obtainCallsInside(I, Block.Entry, Block.Exit , OI, AnalyzedFunctions
                    , CallList);
                //See if it exists a call in the vector that is not inside in a task
                bool AllInsideTask=true;
                SmallVector<TaskInfo ,4> TaskWithCall;
                for(Instruction *Call : CallList){
                    bool InsideTask=false;
                    for (auto ThisTask : FI.TaskFuncInfo.PostOrder) {
                        if ((OI.dominates(ThisTask.Entry, Call) && !OI.dominates(
                            ThisTask.Exit, Call))){
                            TaskWithCall.push_back(ThisTask);
                            InsideTask=true;
                            break;
                        }
                    }

                    if(!InsideTask){
                        AllInsideTask=false;
                        break;
                    }
                }
                //If all are in tasks, check if all the tasks are syncronized with
                    the main
                if(AllInsideTask){
                    bool AllSynchronized= true;
                    for( auto ThisTask: TaskWithCall){
                        if(!existsTaskDependency(ThisTask, SelectedTask)){
                        AllSynchronized=false;
                        break;
                        }
                    }
                    // If they are syncronized, erase the last values added, else
                        mark ConcurrentUseInTask as true
                    if(AllSynchronized){
                        //Restore
                        UsedInConcurrent =PreviousUsedInConcurrent;
                        for(int i=0; i< numAdditions; i++)
                            UsedInConcurrentValues.pop_back();
                    }
                    else{
                        for(int i=0; i< numAdditions; i++)
                            UsedInConcurrentValues[UsedInConcurrentValues.size()-1-i].
                                ConcurrentUseInTask=true;
                    }
                }
            }
        }
    }
} else {
    // In case the use is not an instruction, check for its uses again
    for (User *UNew : U->users()) {
        if (Instruction *I = dyn_cast<Instruction>(UNew)) {
            SmallPtrSet<Function *, 10> AnalyzedFunctions;
            int TaskContainsCall=UseIsInExternalCall(I, SelectedTask.Entry,
                SelectedTask.Exit, OI, AnalyzedFunctions);

            if ((OI.dominates(SelectedTask.Entry, I) && !OI.dominates(SelectedTask.
                Exit, I)) || TaskContainsCall) {
                AnalyzedFunctions.clear();
                valueInInstruction(I, U, UsedInTask, false, &AnalyzedFunctions,
                    UsedInTaskValues, true, dummy, BAA, NONE, true);
            }
```

```
                for (ConcurrentBlock &Block : SelectedTask.ConcurrentBlocks) {
                   AnalyzedFunctions.clear();
                   int TaskContainsCall=UseIsInExternalCall(I, Block.Entry, Block.Exit
                        ,OI, AnalyzedFunctions);

                   // Use is inside a concurrent block
                   if ((OI.dominates(Block.Entry, I) && !OI.dominates(Block.Exit, I))
                        || I == Block.Entry || I == Block.Exit || TaskContainsCall) {
                      AnalyzedFunctions.clear();
                      int previousSize = UsedInConcurrentValues.size();
                      VarUse PreviousUsedInConcurrent = UsedInConcurrent;
                      valueInInstruction(I, U, UsedInConcurrent, false, &
                          AnalyzedFunctions, UsedInConcurrentValues, true, dummy, BAA,
                          NONE, true);

                      int numAdditions= UsedInConcurrentValues.size() - previousSize;
                      // Check if the use was inside a task or not
                      if(Block.Entry->getFunction() == I->getFunction()){
                         for (auto ThisTask : FI.TaskFuncInfo.PostOrder) {
                            if ((OI.dominates(ThisTask.Entry, I) && !OI.dominates(
                                 ThisTask.Exit, I))) {
                               //Check if exists a taskdependency, if it exists we
                                    remove the use since is no concurrent anymore
                               if(isPotentiallyReachable(SelectedTask.Exit, ThisTask.
                                    Entry) && !existsTaskDependency(ThisTask,
                                    SelectedTask)){
                                  UsedInConcurrentValues.back().ConcurrentUseInTask =
                                       true;
                                  break;
                               }
                               else{
                                  UsedInConcurrentValues.pop_back();
                                  break;
                               }
                            }
                         }
                      }
                      else{
                         //Obtain a vector of all calls inside the concurrent block
                         SmallVector<Instruction *, 4> CallList;
                         AnalyzedFunctions.clear();
                         obtainCallsInside(I, Block.Entry, Block.Exit ,OI,
                             AnalyzedFunctions, CallList);
                         //See if it exists a call in the vector that is not inside in
                             a task
                         bool AllInsideTask=true;
                         SmallVector<TaskInfo ,4> TaskWithCall;
                         for(Instruction *Call : CallList){
                            bool InsideTask=false;
                            for (auto ThisTask : FI.TaskFuncInfo.PostOrder) {
                               if ((OI.dominates(ThisTask.Entry, Call) && !OI.dominates
                                    (ThisTask.Exit, Call))){
                                  TaskWithCall.push_back(ThisTask);
                                  InsideTask=true;
                                  break;
                               }
                            }
                            if(!InsideTask){
                               AllInsideTask=false;
                               break;
                            }
                         }
                         //If all are in tasks, check if all the tasks are syncronized
                              with the main
                         if(AllInsideTask){
                            bool AllSynchronized= true;
                            for( auto ThisTask: TaskWithCall){
                               if(!existsTaskDependency(ThisTask, SelectedTask)){
                                  AllSynchronized=false;
                                  break;
                               }
                            }
```

```
                                // If they are syncronized, erase the last values added,
                                    else mark ConcurrentUseInTask as true
                                if(AllSynchronized){
                                    //Restore
                                    UsedInConcurrent =PreviousUsedInConcurrent;
                                    for(int i=0; i< numAdditions; i++)
                                      UsedInConcurrentValues.pop_back();
                                }
                                else{
                                    for(int i=0; i< numAdditions; i++)
                                        UsedInConcurrentValues[UsedInConcurrentValues.size()
                                            -1-i].ConcurrentUseInTask=true;
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    // Auxiliar vectors for storing already visited functions and blocks
    SmallPtrSet<BasicBlock *, 10> AnalizedBasicBlocks;
    SmallPtrSet<Function *, 10> AnalyzedFunctions;

    // Analyze the first use inside the task, to know if it is a read or a write
    for (User *U : Var->users()) {
       if (!dyn_cast<Instruction>(U)) {
          // Case the use is not an instruction
          for (auto &AccessToCheck : UsedInTaskValues) {
             AnalizedBasicBlocks.clear();
             AnalyzedFunctions.clear();
             analyzeFirstTaskUse(SelectedTask.Entry->getParent(), U, SelectedTask.Exit,
                   SelectedTask.Entry, &AnalizedBasicBlocks, &AnalyzedFunctions,
                  AccessToCheck, BAA, BEFORE_ENTRY, true);
          }
       }
    }

    // Case the use is an instruction
    for (auto &AccessToCheck : UsedInTaskValues) {
       AnalizedBasicBlocks.clear();
       AnalyzedFunctions.clear();
       analyzeFirstTaskUse(SelectedTask.Entry->getParent(), Var, SelectedTask.Exit,
           SelectedTask.Entry, &AnalizedBasicBlocks, &AnalyzedFunctions, AccessToCheck,
            BAA , BEFORE_ENTRY, true);
    }

    // Analyze variable is alive after next sync
    for (SyncInfo *nextSync : SelectedTask.PostSyncs)
       for (auto &AccessToCheck : UsedInTaskValues) {
          AnalizedBasicBlocks.clear();
          AnalyzedFunctions.clear();
          for (auto U : Var->users())
             if (!dyn_cast<Instruction>(U))
                analyzeFirstTaskUse(nextSync->Sync->getParent(), U, nullptr, nextSync->
                     Sync, &AnalizedBasicBlocks, &AnalyzedFunctions, AccessToCheck, BAA,
                      AFTER_SYNC, true);
          AnalizedBasicBlocks.clear();
          AnalyzedFunctions.clear();
          analyzeFirstTaskUse(nextSync->Sync->getParent(), Var, nullptr, nextSync->Sync
              , &AnalizedBasicBlocks, &AnalyzedFunctions, AccessToCheck, BAA,
               AFTER_SYNC, true);
       }

    // Analyze variable is alive after task exit
    for (auto &SyncPoint : SelectedTask.PostSyncs) {
       std::vector<std::vector<BasicBlock *>> FinalPaths;
       getPaths(SelectedTask.Exit, SyncPoint->Sync, FinalPaths, OI, false);
       for (auto &Node : FinalPaths) {
          for (auto &BB : Node) {
```

```
            for (auto &AccessToCheck : UsedInTaskValues) {
               AnalizedBasicBlocks.clear();
               AnalyzedFunctions.clear();
               for (auto U : Var->users())
                  if (!dyn_cast<Instruction>(U))
                     analyzeFirstTaskUse(BB, U, SyncPoint->Sync, SelectedTask.Exit, &
                         AnalizedBasicBlocks, &AnalyzedFunctions, AccessToCheck, BAA,
                         AFTER_EXIT, false);
               AnalizedBasicBlocks.clear();
               AnalyzedFunctions.clear();
               analyzeFirstTaskUse(BB, Var, SyncPoint->Sync, SelectedTask.Exit, &
                   AnalizedBasicBlocks, &AnalyzedFunctions, AccessToCheck, BAA,
                   AFTER_EXIT, false);
            }
         }
      }
   }

   // Check if the variable is global
   if (dyn_cast<GlobalValue>(Var))
      IsGlobalVariable = true;

   SmallVector<ValueAccess, 4> DefinitiveUsedValues;
   SmallVector<ValueAccess, 4> ProcessedValueAccess;
   SmallVector<ValueAccess, 4> CopyUsedInTaskValues = UsedInTaskValues;
   // Eliminate uses of same memory address
   for (ValueAccess FirstTaskMemUse : UsedInTaskValues) {
      bool Already = false;
      for (auto &VA : ProcessedValueAccess) {
         if ((FirstTaskMemUse.MemLoc.Ptr != nullptr && VA.MemLoc.Ptr != nullptr) && ((
             FirstTaskMemUse.I->getFunction() != VA.I->getFunction()) || (BAA.alias(
             FirstTaskMemUse.MemLoc, VA.MemLoc) != NoAlias))) {
            Already = true;
            break;
         }
      }
      if (Already)
         continue;
      ProcessedValueAccess.push_back(FirstTaskMemUse);

      SmallVector<VarUse, 4> Scopes;
      Scopes.push_back(FirstTaskMemUse.Use);

      for (ValueAccess SecondTaskMemUse : UsedInTaskValues) {
         if (FirstTaskMemUse.I != SecondTaskMemUse.I && FirstTaskMemUse.MemLoc.Ptr !=
             nullptr && SecondTaskMemUse.MemLoc.Ptr != nullptr) {
            if ((FirstTaskMemUse.I->getFunction() != SecondTaskMemUse.I->getFunction()
                ) || (BAA.alias(FirstTaskMemUse.MemLoc, SecondTaskMemUse.MemLoc) !=
                NoAlias)) {
               Scopes.push_back(SecondTaskMemUse.Use);
            }
         }
      }

      VarUse FinalUse = NOT_USED;
      for (auto TypeOfUse : Scopes) {
         if (TypeOfUse == UNKNOWN) {
            FinalUse = TypeOfUse;
            break;
         }
         if (FinalUse == NOT_USED || (TypeOfUse == WRITTEN && FinalUse == READED))
            FinalUse = TypeOfUse;
      }
      FirstTaskMemUse.Use = FinalUse;
      DefinitiveUsedValues.push_back(FirstTaskMemUse);
   }

   UsedInTaskValues = DefinitiveUsedValues;

   DefinitiveUsedValues.clear();
   ProcessedValueAccess.clear();
```

```
1        // Eliminate uses of same memory address
2        for (ValueAccess &FirstConcurrentMemUse : UsedInConcurrentValues) {
3           bool Already = false;
4           for (auto &VA : ProcessedValueAccess) {
5              if ((FirstConcurrentMemUse.MemLoc.Ptr != nullptr && VA.MemLoc.Ptr != nullptr)
                     && ((FirstConcurrentMemUse.I->getFunction() != VA.I->getFunction()) || (
                    BAA.alias(FirstConcurrentMemUse.MemLoc, VA.MemLoc) != NoAlias))) {
6                 Already = true;
7                 break;
8              }
9           }
10          if (Already)
11             continue;
12          ProcessedValueAccess.push_back(FirstConcurrentMemUse);
13
14          SmallVector<VarUse, 4> Scopes;
15          Scopes.push_back(FirstConcurrentMemUse.Use);
16          for (ValueAccess SecondConcurrentMemUse : UsedInConcurrentValues) {
17             if (FirstConcurrentMemUse.I != SecondConcurrentMemUse.I &&
                    FirstConcurrentMemUse.MemLoc.Ptr != nullptr && SecondConcurrentMemUse.
                    MemLoc.Ptr != nullptr) {
18                if ((FirstConcurrentMemUse.I->getFunction() != SecondConcurrentMemUse.I->
                       getFunction()) || (BAA.alias(FirstConcurrentMemUse.MemLoc,
                       SecondConcurrentMemUse.MemLoc) != NoAlias)) {
19                   Scopes.push_back(SecondConcurrentMemUse.Use);
20                   //Required to know if there is a concurrent use outside the task, since
                          we are eliminating it
21                   if(FirstConcurrentMemUse.ConcurrentUseInTask == true &&
                          SecondConcurrentMemUse.ConcurrentUseInTask == false){
22                      FirstConcurrentMemUse.ConcurrentUseInTask= false;
23                   }
24                }
25             }
26          }
27
28          VarUse FinalUse = NOT_USED;
29          for (auto TypeOfUse : Scopes) {
30             if (TypeOfUse == UNKNOWN) {
31                FinalUse = TypeOfUse;
32                break;
33             }
34             if (FinalUse == NOT_USED || (TypeOfUse == WRITTEN && FinalUse == READED))
35                FinalUse = TypeOfUse;
36          }
37          FirstConcurrentMemUse.Use = FinalUse;
38          DefinitiveUsedValues.push_back(FirstConcurrentMemUse);
39       }
40       UsedInConcurrentValues = DefinitiveUsedValues;
41
42       // Analyze variables that satisfy the conditions
43       if (UsedInTask != UNKNOWN && UsedInTask != NOT_USED && UsedInConcurrent != UNKNOWN)
            {
44          SmallVector<MemoryAccessDescription, 4> AllUses;
45          SmallVector<DSAValue, 4> DSA;
46
47          for (ValueAccess TaskMemUse : UsedInTaskValues) {
48             VarUse LocalUsedInTask = TaskMemUse.Use;
49             VarUse LocalUsedInConcurrent = NOT_USED;
50             VarUse FirstTaskUse = TaskMemUse.FirstTaskUse;
51             bool IsAliveAfterNextSync = TaskMemUse.IsAliveAfterNextSync;
52             bool Found = false;
53             for (ValueAccess ConcurrentMemUse : UsedInConcurrentValues) {
54                if ((TaskMemUse.I->getFunction() != ConcurrentMemUse.I->getFunction()) ||
                       BAA.alias(TaskMemUse.MemLoc, ConcurrentMemUse.MemLoc) != NoAlias) {
55                   Found = true;
56                   LocalUsedInConcurrent = ConcurrentMemUse.Use;
57                   AllUses.push_back({LocalUsedInTask, LocalUsedInConcurrent, FirstTaskUse
                          , IsAliveAfterNextSync, ConcurrentMemUse.ConcurrentUseInTask});
58                }
59             }
```

```
            if (!Found) {
               AllUses.push_back({LocalUsedInTask, LocalUsedInConcurrent, FirstTaskUse,
                   IsAliveAfterNextSync, false});
            }
         }
         for (auto ThisUse : AllUses) {
            VarUse TaskUse = ThisUse.UsedInTask;
            VarUse ConcurrentUse = ThisUse.UsedInConcurrent;
            VarUse FirstInTask = ThisUse.FirstTaskUse;
            bool IsAliveAfterNextSync = ThisUse.IsAliveAfterNextSync;
            bool ConcurrentUseInTask = ThisUse.ConcurrentUseInTask;

            if (ConcurrentUse == NOT_USED) {
               if (TaskUse == READED)
                  DSA.push_back(SHARED_OR_FIRSTPRIVATE);
               else if (TaskUse == WRITTEN) {
                  if (IsGlobalVariable || IsAliveAfterNextSync)
                     DSA.push_back(SHARED);
                  else if (FirstInTask == WRITTEN)
                     DSA.push_back(PRIVATE);
                  else if (FirstInTask == READED || FirstInTask == UNKNOWN)
                     DSA.push_back(SHARED_OR_FIRSTPRIVATE);
               }
            } else {
               if (TaskUse == READED && ConcurrentUse == READED)
                  DSA.push_back(SHARED_OR_FIRSTPRIVATE);
               else if (TaskUse == WRITTEN || ConcurrentUse == WRITTEN) {
                  // TODO: No Data Race (critical section)
                  if (AutoDeps) {
                     if (IsAliveAfterNextSync && !ConcurrentUseInTask)
                        DSA.push_back(UNDEF);
                     else if (!ConcurrentUseInTask) {
                        if (FirstInTask == WRITTEN)
                           DSA.push_back(RACEPRIVATE);
                        else
                           DSA.push_back(RACEFIRSTPRIVATE);
                     } else {
                        dbgs()<< "Possible race condition with other tasks, running
                            Autodeps \n";
                        AutoDepsActivated=true;
                        DSA.push_back(SHARED);
                     }
                  } else {
                     if (IsAliveAfterNextSync)
                        DSA.push_back(UNDEF);
                     else if (FirstInTask == WRITTEN)
                        DSA.push_back(RACEPRIVATE);
                     else
                        DSA.push_back(RACEFIRSTPRIVATE);
                  }
               }
            }
         }

         if (IsComposited) {
            SmallVector<std::pair<DSAValue, DSAValue>, 4> DSAPairs;

            for (int i = 0; i < (int)DSA.size(); i++)
               for (int j = i; j < (int)DSA.size(); j++)
                  DSAPairs.push_back({DSA[i], DSA[j]});

            bool AllEqual = true;
            bool AllRaces = true;
            bool ExistsUndefined = false;
            bool ConditionA = false;
            bool ConditionB = true;
            bool ConditionC = true;
            bool ConditionC1 = true;
            bool ConditionD = false;

            for (auto Pair : DSAPairs) {
               if (Pair.first != Pair.second)
                  AllEqual = false;
```

```
            if (Pair.first == UNDEF || Pair.second == UNDEF)
               ExistsUndefined = true;

            if (((Pair.first == RACEFIRSTPRIVATE || Pair.first == RACEPRIVATE) && Pair
                .second == SHARED) || (((Pair.second == RACEFIRSTPRIVATE || Pair.
                second == RACEPRIVATE) && Pair.first == SHARED)))
               ConditionA = true;

            if (!((Pair.first == SHARED_OR_FIRSTPRIVATE || Pair.first == SHARED) && (
                Pair.second == SHARED_OR_FIRSTPRIVATE || Pair.second == SHARED)))
               ConditionB = false;

            if (!((Pair.first == RACEPRIVATE || Pair.first == PRIVATE) && (Pair.second
                 == RACEPRIVATE || Pair.second == PRIVATE)))
               ConditionC = false;

            if (!((Pair.first == RACEFIRSTPRIVATE || Pair.first == PRIVATE) && (Pair.
                second == RACEFIRSTPRIVATE ||  Pair.second == PRIVATE)))
               ConditionC1 = false;

            if ((Pair.first == SHARED_OR_FIRSTPRIVATE && (Pair.second == RACEPRIVATE
                || Pair.second == RACEFIRSTPRIVATE || Pair.second == PRIVATE)) || (
                Pair.second == SHARED_OR_FIRSTPRIVATE && (Pair.first == RACEPRIVATE ||
                 Pair.first == RACEFIRSTPRIVATE || Pair.first == PRIVATE)))
               ConditionD = true;

            if (!((Pair.first == RACEPRIVATE || Pair.first == RACEFIRSTPRIVATE) && (
                Pair.second == RACEPRIVATE || Pair.second == RACEFIRSTPRIVATE)))
               AllRaces = false;
         }

         if (AllEqual || DSA.size() == 1)
            CorrectedDSA = DSA[0];
         else if (ExistsUndefined || ConditionA)
            CorrectedDSA = UNDEF;
         else if (ConditionB)
            CorrectedDSA = SHARED;
         else if (ConditionC)
            CorrectedDSA = PRIVATE;
         else if (ConditionD || ConditionC1 || AllRaces)
            CorrectedDSA = FIRSTPRIVATE;
         else
            assert(false && "Error: DSA not found");
      } else {
         DSAValue FinalDSA = UNINITIALIZED;
         for (auto ThisDSA : DSA) {
            if (ThisDSA == UNDEF) {
               FinalDSA = ThisDSA;
               break;
            }
            if (ThisDSA == FIRSTPRIVATE || ThisDSA == RACEFIRSTPRIVATE)
               FinalDSA = ThisDSA;
            if ((ThisDSA == PRIVATE || ThisDSA == RACEPRIVATE) && (FinalDSA ==
                SHARED_OR_FIRSTPRIVATE || FinalDSA == SHARED || FinalDSA ==
                UNINITIALIZED))
               FinalDSA = ThisDSA;
            if (ThisDSA == SHARED && (FinalDSA == UNINITIALIZED || FinalDSA ==
                SHARED_OR_FIRSTPRIVATE))
               FinalDSA = ThisDSA;
            if (ThisDSA == SHARED_OR_FIRSTPRIVATE && (FinalDSA == UNINITIALIZED))
               FinalDSA = ThisDSA;
         }
         CorrectedDSA = FinalDSA;
         assert(CorrectedDSA != UNINITIALIZED && "Error: DSA not found");
      }

      if (CorrectedDSA == RACEPRIVATE){
         CorrectedDSA = PRIVATE;
         dbgs() << "Race condition detected, can not be solved with autodeps,
             privatizing the variable \n";
      }
```

```
        else if (CorrectedDSA == RACEFIRSTPRIVATE){
            dbgs() << "Race condition detected, can not be solved with autodeps,
                firstprivatizing the variable \n";
            CorrectedDSA = FIRSTPRIVATE;
        }
    }
    bool isUnknown = false;
    if (CorrectedDSA == UNDEF) {
        if (UsedInTask != NOT_USED) {
            isUnknown = true;
            numUnknown++;
        }
        CorrectedDSA = OriginalDSA;
    } else if (CorrectedDSA == SHARED)
        numShared++;
    else if (CorrectedDSA == PRIVATE)
        numPrivate++;
    else if (CorrectedDSA == FIRSTPRIVATE)
        numFirstPrivate++;
    else if (CorrectedDSA == SHARED_OR_FIRSTPRIVATE) {
        if (IsComposited || IsPointer) {
            numShared++;
        } else {
            numFirstPrivate++;
        }
    }

    if (CorrectedDSA == SHARED)
        SelectedTask.DSAInfo.Shared.insert(Var);
    else if (CorrectedDSA == PRIVATE)
        SelectedTask.DSAInfo.Private.insert(Var);
    else if (CorrectedDSA == FIRSTPRIVATE)
        SelectedTask.DSAInfo.Firstprivate.insert(Var);
    else if (CorrectedDSA == SHARED_OR_FIRSTPRIVATE) {
        if (IsComposited || IsPointer) {
            SelectedTask.DSAInfo.Shared.insert(Var);
            CorrectedDSA = SHARED;
        } else {
            SelectedTask.DSAInfo.Firstprivate.insert(Var);
            CorrectedDSA = FIRSTPRIVATE;
        }
    }

    if (PrintVerboseLevel == PV_AutoScoping) {
        Var->printAsOperand(dbgs(), false);
        if (isUnknown)
            dbgs() << " detected scope: UNKNOWN";
        else
            dbgs() << " detected scope: " << DSAToString[CorrectedDSA];

        if (CorrectedDSA != OriginalDSA)
            dbgs() << ", modified original scope was: " << DSAToString[OriginalDSA] << "\
                n";
    }

    if (AutoDeps && AutoDepsActivated && CorrectedDSA == SHARED) {
        int VarDependency = DEP_NONE;
        for (auto VarUse : CopyUsedInTaskValues) {
            bool UsedBeforeEntry = false;
            bool UsedAfterExit = VarUse.IsAliveAfterExit;
            bool ConditionA = true;
            if (VarUse.IsAliveBeforeEntry) {
                UsedBeforeEntry = true;
                bool UsedInConcurrent = false;
                bool UsedInATask = false;
                for (auto ConcurrentUse : UsedInConcurrentValues)
                    if (ConcurrentUse.I == VarUse.I)
                        UsedInConcurrent = true;

                // TODO ONLY TASK at same level
                for (auto ThisTask : FI.TaskFuncInfo.PostOrder) {
                    if (OI.dominates(ThisTask.Entry, VarUse.I) && !OI.dominates(ThisTask.
                        Exit, VarUse.I))
                        UsedInATask = true;
                }
```

```
            if (!UsedInATask && UsedInConcurrent) {
               ConditionA = false;
            }
         }
         if (UsedBeforeEntry && UsedAfterExit && UsedInTask==WRITTEN) {
            VarDependency= DEP_INOUT;
         } else if (UsedAfterExit && UsedInTask==WRITTEN) {
            if(VarDependency == DEP_NONE)
               VarDependency = DEP_OUT;
            else if(VarDependency==DEP_IN)
               VarDependency= DEP_INOUT;
         } else if (UsedBeforeEntry) {
            if (ConditionA){
               if(VarDependency == DEP_NONE)
                  VarDependency = DEP_IN;
               else if(VarDependency==DEP_OUT)
                  VarDependency=DEP_INOUT;
            }
         }
      }

      if(VarDependency != DEP_NONE){
         bool correctDEP= false;
         if(VarDependency== DEP_IN){
            for (DependInfo In : SelectedTask.DependsInfo.Ins){
               if(In.Base == Var){
                  correctDEP = true;
                  break;
               }
            }
         }
         else if(VarDependency== DEP_OUT){
            for (DependInfo Out : SelectedTask.DependsInfo.Outs){
               if(Out.Base == Var){
                  correctDEP = true;
                  break;
               }
            }
         }
         else if(VarDependency== DEP_INOUT){
            for (DependInfo Inout : SelectedTask.DependsInfo.Inouts){
               if(Inout.Base == Var){
                  correctDEP = true;
                  break;
               }
            }
         }
         if(!correctDEP){
            dbgs() << "\033[1;31mPossible␣ERROR:␣" ;
            Var->printAsOperand(dbgs(), false);
            dbgs() << "␣should␣be␣" << VarDependencyToString[VarDependency] << "␣
                \033[0m\n";
         }
         else{
            dbgs() << "\033[1;32mDependencies␣seems␣OK!␣\033[0m\n";
         }
      }
   }
   else if(!AutoDepsActivated){
      bool depExists= false;

      for (DependInfo In : SelectedTask.DependsInfo.Ins){
         if(In.Base == Var){
            depExists = true;
            break;
         }
      }
      if(!depExists)
         for (DependInfo Out : SelectedTask.DependsInfo.Outs){
            if(Out.Base == Var){
               depExists = true;
               break;
            }
         }
```

```
            if(!depExists)
               for (DependInfo Inout : SelectedTask.DependsInfo.Inouts){
                  if(Inout.Base == Var){
                     depExists = true;
                     break;
                  }
               }
            if(depExists){
               dbgs() << "\033[1;35mPossible␣Warning:␣";
               Var->printAsOperand(dbgs(), false);
               dbgs() << "␣should␣not␣be␣a␣dependency␣\033[0m\n";
            }
         }
         if (!DisableAutorelease) {
            // Calculate automatic release clauses for outs
            for (DependInfo &OutDepend : SelectedTask.DependsInfo.Outs)
               if (isSameVar(OutDepend.Base, Var)) {
                  // Only shared variables can be released
                  bool IsShared = false;
                  for (auto SharedVar : SelectedTask.DSAInfo.Shared) {
                     if (isSameVar(OutDepend.Base, SharedVar))
                        IsShared = true;
                  }

                  if (IsShared)
                     getReleasePoints(SelectedTask, CopyUsedInTaskValues, OutDepend, OI, FI.
                          ReleaseFuncInfo.PostOrder, OUT, LI, LA, BAA, SE, AC, DT, F);
               }

            // Calculate automatic release clasues for inouts
            for (DependInfo &InOutDepend : SelectedTask.DependsInfo.Inouts)
               if (isSameVar(InOutDepend.Base, Var)) {
                  // Only shared variables can be released
                  bool IsShared = false;
                  for (auto SharedVar : SelectedTask.DSAInfo.Shared) {
                     if (isSameVar(InOutDepend.Base, SharedVar))
                     IsShared = true;
                  }

                  if (IsShared)
                     getReleasePoints(SelectedTask, CopyUsedInTaskValues, InOutDepend, OI,
                          FI.ReleaseFuncInfo.PostOrder, INOUT, LI, LA, BAA, SE, AC, DT, F);
               }
         }
      }

      LLVM_DEBUG(dbgs() << "Num␣SHARED␣" << numShared << "␣Num␣PRIVATE␣"
                        << numPrivate << "␣Num␣FIRSTPRIVATE␣" << numFirstPrivate
                        << "␣Num␣UNDEF␣" << numUnknown << "␣\n");

      GnumShared += numShared;
      GnumPrivate += numPrivate;
      GnumFirstPrivate += numFirstPrivate;
      GnumUnknown += numUnknown;
   }

   LLVM_DEBUG(dbgs() << "\n");
   if (PrintVerboseLevel == PV_AutoRelease)
      for (ReleaseInfo ReleasePoint : FI.ReleaseFuncInfo.PostOrder) {
         dbgs() << "[Release␣point]␣" << ReleasePoint.I->getParent()->getName() << "␣\n";
      }

   if ((GnumShared + GnumPrivate + GnumUnknown + GnumFirstPrivate) > 0)
      LLVM_DEBUG(dbgs() << "GLOBAL␣Num␣SHARED␣" << GnumShared << "␣Num␣PRIVATE␣" <<
           GnumPrivate << "␣Num␣FIRSTPRIVATE␣" << GnumFirstPrivate << "␣Num␣UNDEF␣" <<
           GnumUnknown << "␣\n");
```

Figure 33: Modifications targeting OpenMP correctness analysis in the LLVM compiler.