Framework for the Analysis and Configuration of Real-Time OpenMP Applications

Tiago Carvalho, Luis Miguel Pinho, Mohammad Samadi
Polytechnic Institute of Porto
Porto, Portugal
{tdc,lmp,mmasa}@isep.ipp.pt

Sara Royuela, Adrian Munera, Eduardo Quiñones
Barcelona Supercomputing Center
Barcelona, Spain
{sara.royuela,adrian.munera,eduardo.quinones}@bsc.es

***Abstract*—High-performance cyber-physical applications impose several requirements with respect to performance, functional correctness and non-functional aspects. Nowadays, the design of these systems usually follows a model-driven approach, where models generate executable applications, usually with an automated approach. As these applications might execute in different parallel environments, their behavior becomes very hard to predict, and making the verification of non-functional requirements complicated. In this regard, it is crucial to analyse and understand the impact that the mapping and scheduling of computation have on the real-time response of the applications. In fact, different strategies in these steps of the parallel orchestration may produce significantly different interference, leading to different timing behaviour.**

Tuning the application parameters and the system configuration proves to be one of the most fitting solutions. The design space can however be very cumbersome for a developer to test manually all combinations of application and system configurations. This paper presents a methodology and a toolset to profile, analyse, and configure the timing behaviour of high-performance cyber-physical applications and the target platforms. The methodology leverages on the possibility of generating a task dependency graph representing the parallel computation to evaluate, through measurements, different mapping configurations and select the one that minimizes response time.

***Index Terms*—Cyber-Physical Systems, Real-time, Timing Analysis, Task-to-thread Mapping**

I. Introduction

Developing cyber-physical systems is challenging due to the increasing processing requirements of advanced applications and the stringent non-functional requirements caused by the interaction with the physical world. In this context, the AMPERE project [1] designed an ecosystem targeting the correct-by-construction engineering of cyber-physical applications on parallel heterogeneous platforms. The ecosystem generates task-based parallel OpenMP applications from AMALTHEA [2] models, including mechanisms to offload computation to accelerators. This trait allows for defining function specializations based on conditions like the type of processor, e.g., CPU or GPU, hence fostering the deployment of multiple versions of the same application. Different combinations might cause significant variations in the performance of the system, while the potentially high complexity of such a hybrid environment complicates the selection of the best combination.

This work has been co-funded by the European commission through the AMPERE (H2020 grant agreement N° 745601) project.

The performance of the application and the guaranteed response time depends on several factors including the combination of variants, the dependencies among functionalities and so the opportunities for parallelism, the mapping and scheduling algorithms, and the configuration of the target platform (e.g., CPU frequency). All these factors make it difficult to easily provide the most suitable configuration, for each criterion, for a given program, in a given platform.

This paper presents the component in the AMPERE framework that analyses real-time OpenMP applications and provides fitting configurations to execute in a given platform with the objective of delivering the best-guaranteed performance to the applications. The component is integrated with the general AMPERE workflow, but it can also be used as a stand-alone tool to analyse any task-based OpenMP application.

II. The AMPERE ecosystem

The AMPERE project [1] addresses the development of cyber-physical systems (of systems) by leveraging the capabilities of parallel heterogeneous platforms to cope with the demands of increasingly high-performance applications like autonomous vehicles. The main goal is to provide an ecosystem able to support the complete development flow, from the applications' models to the executables, as well as the execution environment, i.e., runtimes, operating systems and hypervisors, in the considered heterogeneous platforms.

This paper does not intend to provide a complete description of the AMPERE ecosystem (interested readers are referred to [1] and [3]), but rather focuses on the flow to determine the application and platform configurations for providing the required real-time guarantees. This section thus provides a brief description of the AMPERE concepts and tools required for a complete understating of this flow.

A. Nomenclature in AMPERE

The AMPERE ecosystem is composed of three different phases: (1) the *modeling* phase, where use cases are described through AMALTHEA models; (2) the *parallelism* phase, where models are transformed into parallel OpenMP code; and (3) the analysis and optimisation phase, which leverages the task dependency graph (TDG) generated at compile-time for multi-criteria optimisation. These phases and the different nomenclature related to them are summarized in Table I.

TABLE I
NOMENCLATURE OF THE AMPERE ECOSYSTEM.

Phase	Tool	File Extension	Process	Functionality
Modeling	AMALTHEA	model.amxmi	(AMALTHEA) Task	Runnable
Parallelism	OpenMP	code.c/.cpp	Parallel region	(OpenMP) Task
Optimization	Multi-criteria	tdg.json	TDG	Node

In the AMPERE ecosystem, AMALTHEA tasks are transformed into OpenMP parallel regions, with runnables being executed as OpenMP tasks. Data dependencies between runnables (expressed as labels in the model) are mapped into data dependencies between OpenMP tasks, thus effectively allowing to safely parallelize the runnables within an AMALTHEA task by avoiding race conditions. The ecosystem further exploits the task dependency graph (TDG) that represents the parallel execution of an AMALTHEA task, where nodes are OpenMP tasks (or runnables) and edges are dependencies among them.

B. From AMALTHEA to OpenMP: Synthetic Load Generator

The Synthetic Load Generator (SLG) [3] is a code generator capable of transforming AMALTHEA models into OpenMP/C/C++ code. AMALTHEA tasks and runnables are transformed into C functions. Then, the code within the tasks, i.e., the sequence of calls to the runnables contained in the AMALTHEA task, is parallelized using OpenMP tasks. Moreover, when runnables have defined different specializations, the SLG can generate several applications, as much as the Cartesian product of all sets of runnable specializations.

C. The Task Dependency Graph

The work in AMPERE considers Task Dependency Graphs (TDGs) [4], which are a simplified form of directed acyclic graphs (DAGs) that consider only sibling nodes (i.e., no nested graphs) and the dependencies among them. The TDG is a simple structure where each node (an OpenMP task) only includes the *in* and *out* dependencies with other tasks, which are extracted from the depend clauses defined in the directives annotating the OpenMP tasks. This representation enables (1) the analysis and configuration of OpenMP parallel regions, and (2) the use of correctness analysis techniques that allow for verifying the parallelization with respect to the model

D. Compilation with LLVM

The compilation process is performed using an extended version of LLVM [5] developed by the Barcelona Supercomputing Center (BSC). The features include the generation of OpenMP TDGs for user-/model-defined `taskgraph` regions [4] [1]. The support is required at three different levels, i.e., the Clang front-end, the LLVM compiler and the OpenMP runtime. For the purpose of AMPERE, the TDG is generated in two formats: (1) a ".dot" file containing the visual representation of TDG in the parallel region, and (2) a ".cpp" file with the source code of the TDG structure to be used by the runtime. This structure includes several parameters related to the tasks, like the *static_thread* parameter, which adds the possibility of statically mapping OpenMP tasks to threads.

[1]The taskgraph framework for exploiting TDGs is already partially featured in upstream LLVM [6]

E. Performance analysis with Extrae

AMPERE leverages Extrae for the analysis of OpenMP applications. Extrae [7] is a performance monitor tool developed by BSC able to monitor parallel applications and provide performance results at the OpenMP task level. It dynamically profiles the execution of the application and generates a set of files representing the execution trace (i.e., .prv, .pcf, and .row), which include the timestamps of the events captured, like entering or exiting a task and punctual information about performance counters. This allows for computing the execution time of tasks and relate it to other aspects of the system. A Python script has been developed in the context of AMPERE to match the information coming from the DOT-formatted TDG generated by the compiler and the CSV-like format trace generated by Extrae. The script parses and converts the trace generated during the execution of the application into a JSON format (".json" extension). The JSON object organizes the results per OpenMP region (a TDG), and per OpenMP task in that region (a node of the TDG), which facilitates the analysis.

III. OPTIMISATION FLOW

The AMPERE ecosystem enables an optimization flow that targets the analysis and configuration of OpenMP parallel applications with timing as an optimisation goal. The flow analyses and optimises the configuration parameters of the application and the system as to ensure a set of non-functional requirements (defined in the model and passed to the analysis through the TDG). Figure 1 illustrates the optimisation flow, where the multiple cpp files and the profiling configuration file, in blue represent the inputs required by the process, white files are automatically generated artifacts, and green boxes represent the stages engaged in the process.

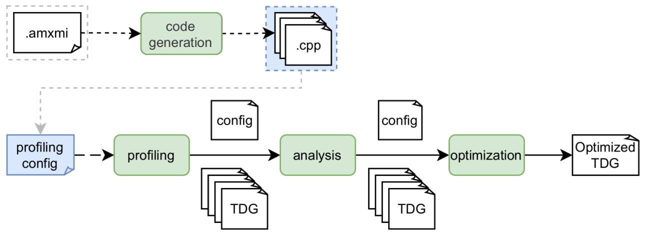

Fig. 1. The real-time optimization flow.

The approach should guarantee that all tasks:

1) execute within their specified deadline (are schedulable),
2) have a more predictable execution, and
3) have an optimized response time (better performance).

The real-time optimization process is divided into four stages: (1) code generation, (2) profiling, (3) timing analysis, and (4) response-time optimization. Two input files are given during the process: an AMALTHEA model and a ”base configuration” file. The former defines the structure and variability of the application while the latter provides the configuration that controls the process. Interestingly, the approach has been developed so that it can be used to optimize any OpenMP application for which the taskgraph is applicable. In this sense, the required input of the optimization tool is the actual OpenMP source code. Therefore, in the context of AMPERE the process starts with code generation, but it can start at the profiling phase if used directly with available OpenMP code.

Most of the communication occurring between the stages of the optimization flow, specifically after the profiling stage, are based on the generated TDG and configuration files. The dashed lines in Figure 1 represent the data moved from the artifacts to the base configuration file, i.e., information concerning chains, periods, and deadlines is propagated from the AMALTHEA model (the .amxmi file), and the generated executable and TDG files from the compilation phase.

The profiling, analysis and optimization artifacts are utilized in a fully automated fashion, not requiring any human interaction. The ”config” file, which is generated during the profiling phase based on the profiling information and the deadline constraints imposed by the application, contains all the necessary data for the automation and it is updated during the analysis and optimization phases. In the end, the process produces a single *TDG.json* file containing the optimized TDGs and the configuration of the system.

IV. The Profiling phase

The *profiling* stage is responsible for providing profiling data from the application (e.g., task execution time, L2 cache misses, instructions per cycle) in different execution scenarios (e.g., clock frequency or energy budget). The profiler deals with multiple versions of the same application, i.e. different combinations of the function specializations, by executing (or simulating) the application a specified number of times (defined in the configuration), with different system configurations. In terms of structure, the multiple versions share the same TDG. The only difference among them is the specialization represented by a given node, more specifically, if the functionality is to be executed in the CPU or the GPU.

A. Profiling Configuration File

The information regarding the version of each specialized functionality in the application and the instructions to setup the target system are defined in the profiling configuration file. Figure 2 illustrates the structure of this file, which is essentially divided into three sections of properties: Application, platform, and optimization.

```
 1  {
 2     "app": {
 3        "tdgs": [{
 4           "id": 1,
 5           "constraints": {
 6              "deadline": 60000000
 7           }
 8        },...],
 9        "variants": [{
10           "dir": "./var1",
11           "build": {
12              "dynamic_mapping": "make",
13              "static_mapping": "make␣static_map"
14           }
15           "uses_gpu": false, ,
16           "iterations": 100
17        },...]
18     },
19     "platform": {
20        "cpu": {
21           "cmd": [
22              "setup_cpu_frequency␣{frequency}",
23                {"OMP_NUM_THREADS": "{threads}"}
24           ],
25           "args": {
26              "frequency": [729600, 1190400, 2265600],
27              "threads": [4,8]
28           },
29        },
30        "gpu": {
31           "cmd": "setup_gpu_frequency␣{frequency}",
32           "args": {
33              "frequency":{
34                 "start": 624750,
35                 "stop": 1377000,
36                 "step":  100000,
37              }
38           }
39        }
40     },
41     "optimization": { ... }
42  }
```

Fig. 2. Sample profiling configuration file with some of the basic properties.

The application section includes information regarding:

– the TDGs, i.e., a list of identifiers (to match it with the traces) and corresponding deadlines (from the represented AMALTHEA task); and
– the variants, i.e., the code version location, if it uses an accelerator device or not, and how a code version is compiled.

For the target system (platform), the configuration specifies how is it possible to reconfigure the system with different parameters. More specifically, this section defines the commands to reconfigure both CPU (line 20 of Figure 2) and GPU (line 30) via the `cmd` attribute, where it is possible to define multiple commands as necessary, to obtain the desired configuration. The most relevant commands are those that reconfigure the system and that affect considerably the performance of the functionalities and, subsequently, the response time of tasks, and include the following aspects:

– the frequency, defined through the script named *setup_cpu_frequency* (for the CPU) or that named *setup_gpu_frequency* (for the GPU) using as argument the value of *frequency*; and
– the number of OpenMP threads (in the CPU section, lines 22 and 23), defined through setting the

OMP_NUM_TRHEADS environment variable using as argument the value of *thread*.

Commands can be parameterized through the {param_name} notation, which can be specified as: A scalar value, an array of values (lines 26 and 27), or a range (line 33). The profiler is responsible for testing all possible combinations of parameter values (a configuration) and execute the program in the different setups.

Finally, the optimization section contains information for the optimization phase, which is discussed in Section VI.

B. Running the Profiler

The profiling phase is composed of one main script and depends on a set of monitoring tools included in the AMPERE ecosystem to provide measurements from executions. The main script starts by compiling each provided version with the compilation commands specified as "dynamic mapping". This will build an executable that runs using the default dynamic task to thread mapping algorithm of OpenMP [8]. In a nutshell, the thread instantiating tasks places the ready ones in its own queue. When work is available, the rest of threads will start stealing tasks from that queue and all threads with tasks in their queues start executing. When a task is finished, the executing thread places any successors with met dependencies in its own queue and continues executing tasks from its queue, if there are, or steals work from other threads, otherwise.

Next, the script iterates over the possible system configurations, which are combinations built from the CPU and GPU parameters. For each system configuration, the script first executes the system configuration commands to reconfigure both CPU and GPU (the latter only for the code variants that use the GPU), and then iterates over each executable a certain number of runs, defined in the configuration as *iterations*.

During the execution of each configuration, the monitoring tool gathers performance data through the execution of the Extrae profiling infrastructure [7]. Extrae collects run-time information regarding execution time and performance counter measurements, and generates the profiling results. The TDG instrumentation script (described in Section II-E) uses these profiling results and the TDGs generated by the compiler (which initially only include OpenMP tasks defined by their ids and their dependencies) to generate a JSON file for the current configuration $\langle code\_version, CPU\_config, GPU\_config \rangle$, annotated with performance results and metadata regarding the configuration data (as seen in Figure 3). This file contains the results of the executions organized per OpenMP parallel region (ergo AMALTHEA task).

As previously stated, one application can have multiple TDGs. All of those TDGs will be present in the TDG.json file. Each TDG (an OpenMP parallel region) contains a list of nodes, or (OpenMP) tasks. Each task of the TDG contains a list of results, where each result represents an iteration/execution of that task. Figure 3 shows an annotated task of a TDG, with one result represented in the image. Each result provides several components, where the most important ones for the

```
1  {
2    "my_app": [
3      {
4        "taskgraph_id": 1,
5        "nodes": {
6          "0": {
7              "ins": [],
8              "outs": ["1","2"],
9              "results": [
10                 {
11                     "execution_total_time": 1297925,
12                     "L1D_CACHE": 24999522,
13                     "L1D_CACHE_REFILL": 5852
14                 }, ...
15             ]
16         }, ...
17       },
18       "metadata": {
19             "cpu": {
20               "frequency": 2265600,
21               "num_threads": 8
22             },
23             "gpu": {}
24       }
25     }
26   ]
27 }
```

Fig. 3. Example of a TDG.json file, showing one TDG specification and one of the tasks of that TDG. The task contains a list of results when executed in the profiling phase. The file also contains information about the system configuration.

timing analysis are the total execution time of the task (e.g. line 11), and the performance counters results during that execution (e.g. cache accesses and refills as in lines 12 and 13 respectively).

At the end of the profiling execution, the tool generates several TDG.json files, one per possible code version and system configuration. These files, together with an intermediate configuration file, are the inputs for the analysis process, which analyses each TDG file to extract metrics, and the optimization phase, which provides the best application and system configuration based on the extracted metrics. The intermediate configuration file is used to control the following stages of the optimization flow. It includes the path to the generated TDG files, and their corresponding platform configuration, and inherits most of the information present in the profiler configuration file, more importantly, the optimization section.

V. Timing Analysis

Following the profiling phase, the flow includes a timing analysis. This analysis is performed over the TDG.json files, providing information at two levels: a per-task analysis and a per TDG analysis.

For each task, the metrics are focused on the execution time and the association of performance counters information. For the execution time, the tool calculates the worst-case execution time (WCET) and the average time.

Regarding performance counters, the ones extracted are specified by the user in the profiling configuration. The performance counters are converted into a set of predefined metrics: average, maximum, minimum, and number of occurrences.

```
1
2  {
3      "taskgraph_id": 1,
4      "nodes": {
5          "0": {
6              "ins": [],
7              "outs": ["1","2"],
8              "metrics": {
9                  "wcet": 1413985,
10                 "avg_time": 1016365,
11                 "L1D_CACHE": {
12                         "avg": 33451818.15,
13                         "max": 43257956,
14                         "min": 15015617,
15                         ...
16                 },
17             }
18         },...
19     }
20     ...
21     "metrics": {
22         "volume": 3445520,
23         "avg_makespan": 2798390,
24         "worst_makespan": 4289395,
25         "max_parallelism":  2
26     }
27 }
```

Fig. 4. Example of annotating metrics in the TDG of Figure 3, and its tasks.

Metrics related to a TDG use both existing information on the TDG and the new metrics calculated per task, more specifically the WCET of each task. A set of metrics can be obtained from the analysis. Since the analysis tool can be used in different stages of the project development, different metrics might be outputted. The set of metrics outputted for the multi-criteria optimization flow is:

- volume
- critical path length
- potential maximum parallelism
- average makespan
- worst-case makespan

The volume is the sum of WCETs of all tasks in the TDG. The critical path length of a task represents the total cost of a path in the TDG that provides the longest path from the source task to the sink task, inclusive.

The potential maximum parallelism is a metric that provides the theoretical maximum parallelism possible in a TDG, disregarding the costs. From all possible, non-dependent siblings, it provides the maximum number of tasks that, in theory, could work in parallel, even if in practice they are not excepted to be executed at the same time.

The `makespan` provides the actual execution time of a TDG, from the start of execution to its end. This is a relevant metric when considering a specific task-to-thread mapping, providing the execution time of the TDG when considering the WCET of all tasks, for a given number of available threads. The critical path length of a TDG can be seen as the lower bound of the makespan the TDG can take.

At the end, the timing analysis annotates the TDG.json file with the calculated metrics, in both the TDGs and their tasks. Figure 4 shows the output of the static timing analysis when Figure 3 was given as input. Task 0 was annotated with its WCET (line 9), the L1 data cache accesses (line 11), and other metrics. The complete TDG was annotated with makespan, volume, and maximum potential parallelism (lines 23 to 25).

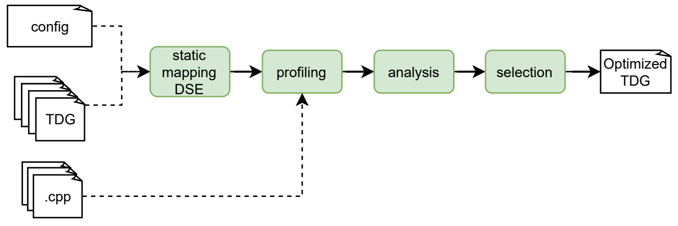

Fig. 5. The optimization flow.

The TDG is now annotated with metrics that will be used in the optimization phase. The tool is also used later in the process to reevaluate the TDGs, taking into account the additional information the optimization phase adds to the graph.

VI. Optimization

The optimization consists of 4 phases: an exploration of the task to thread static mapping algorithms, a reprofiling, a timing analysis, and finally a selection of the best configurations. This flow is represented in Figure 5.

A. *Design-Space Exploration of Static Mapping*

Most of OpenMP implementations support two scheduling algorithms: BFS and WFS. BFS puts a new task into a pool of tasks, so the thread is encountered to continue the execution of the parent task. In contrast, WFS tends to execute new tasks immediately after they are created by the parent's thread, in which the execution of the parent task is suspended.

The use of these default mapping algorithms has some drawbacks, such as: (i) they do not use temporal conditions of the system (e.g., the execution time of parallel tasks) in each phase of the mapping process, they rely on pessimistic analytical techniques for mapping tasks to threads, and the dynamic mapping decision is a problem for having a more predictable execution. The framework uses an approach that takes advantage of temporal information of the application, and a set of mapping algorithms, to define a static mapping for each TDG.

Two concepts are important to define here. The first one is the specification of mapping mechanisms able to outperform the predefined OpenMP algorithms. These can take advantage of the highlighted parameters to provide more fitting mappings. As the mapping algorithms might perform differently for different applications, the second concept is a mechanism to automatically explore different algorithms, select the most fitting one, and annotate the TDGs to statically map tasks to threads. This approach aims to automate the mapping selection, enhance the predictability and robustness of the mapping, minimize the response time of the application, and reduce the running time overhead of the mapping process.

```
{
    "task2thread": "Best-fit",
    "queue": "FIFO",
    "queue_per_thread": true
}

```

Fig. 6. Example of a mapping algorithm specification.

```
1  input: TDG, algorithms, num_threads, deadline
2  output: TDG
3  min_alg = NULL
4  min_t = deadline
5  for each alg in algorithms
6          map = simulate(TDG, alg, num_threads)
7          if map.makespan < min_t
8                  min_alg = map
9                  min_t = map.makespan
10
11 if min_alg == NULL
12         exit(-1)
13
14 apply_map(TDG,map)
```

Fig. 7. Algorithm that simulates the TDG execution with each available algorithm. The result is the TDG annotated with a static mapping.

```
{
    "1": {
        "ins": ["0"],
        "outs": ["3", "4"],
        "metrics": {
            "WCET": 566201,
        },
        "static_thread": 1
    },
}

```

Fig. 8. Example of a task annotated with a static thread.

This phase uses a simulation approach to explore different mapping algorithms, using the heuristic-based mapping approach designed by Gharajeh et al. [9]. This approach separates the mapping into two phases: scheduling and allocation. The scheduling phase maps discovered tasks into OpenMP thread queues, while the allocation phase relates to how a thread decides which tasks to execute next from its queue. Each of these phases has an heuristic associated. From the existing heuristics, the following are highlighted.

For the scheduling phase, the thread queue is selected:

- First-fit: containing the minimum number of tasks;
- Best-fit: with the minimum (accumulated) execution time;
- Recent-fit: with the most recent idle time of threads;
- Optimum-fit: including the maximum response time.

For the allocation phase, the thread selects the task:

- FIFO: that arrived first in the queue;
- Best-fit: having the minimum execution time;
- Optimum-fit: with the maximum response time;
- Multi-criteria: including the shortest execution time and the longest response time.

These algorithms are initially specified in the profiling configuration file and are migrated to the intermediate configuration. The algorithm configuration has three essential properties: a task-to-thread algorithm, the type of queue (i.e. the algorithm that organizes tasks in a queue), and boolean indicating if one queue per thread shall be used. Figure 6 exemplifies the structure of the algorithm specification. The boolean `queue_per_thread` can be used to use different implementations of the algorithm that can deal with a single thread or a queue-per-thread environment. One example of this difference is the different implementations within GOMP and LLVM runtimes [10].

The mapping exploration requires these algorithms, the annotated TDG, the number of threads, and the task deadline. This tool provides an exploration mechanism that iterates the listed algorithms and simulates their execution, based on the TDG structure and WCET of the tasks. It searches for the mapping that provides the best makespan for that TDG. Upon exhausting the algorithms, the tool annotates the TDG with the best mapping and the makespan of that mapping. If the exploration is not able to provide a mapping with a makespan lower than the deadline, then the tool does not annotate the TDG and returns only the lowest makespan possible.

The exploration algorithm is specified in Figure 7. It is composed of a mapping algorithm iterator and a simulator. From the TDG, the simulator uses the task data, its dependencies, and the cost of the task (i.e. the WCET), calculated in the previous step.

The exploration of the algorithms is a traditional loop that iterates all the available algorithms and performs a simulation[2] with that algorithm. The formula used to measure the static mapping is the makespan work of the TDG, which can be calculated by the longest path of the TDG with the provided mapping. Lines 7 and 9 of Figure 7 show the makespan being used to see if the mapping provides an execution time within the deadline, and if it is less than the deadline (and consequently less than the previous algorithm) the selected algorithm is updated. Line 4 contains a simple optimization in the algorithm, that immediately sets the current minimum time to the deadline, reducing the number of comparisons to be done per mapping algorithm.

Statically mapping tasks to threads is done by annotating each task in the TDG with the property `static_thread` with the corresponding thread id, depicted in the mapping provided in the previous steps. Figure 8 shows an example of a task annotated with two properties not present at the first version of the TDG: `WCET` and `static_thread`. This information will be passed on to the following step of the optimization phase.

B. Reprofiling

The objective of this phase is to have measurements more specific to the static mapping defined for each TDG. Since each task is statically mapped to a thread, the obtained

[2]A simulation is performed instead of actual execution since it is intended to use existing measurements to provide a static mapping of tasks to threads.

performance results are much more accurate, considering the expected execution of each task. The reprofiling phase uses the same profiler as defined in Section IV, but instead of using the "dynamic_mapping" compilation, the "static_mapping" compilation is used (see line 13 of Figure 2).

In this phase, each TDG has information about the static mapping and the system configuration in which it was executed previously. The reprofiling phase is then a loop iterating each TDG that:

1) recompiles the corresponding code variant with the provided static mapping;
2) reconfigures the system with the provided configuration;
3) measures the execution of the compiled version;
4) and redefines the TDG with the new results.

C. Final Analysis and Selection

After reprofiling all the TDGs, the timing analysis tool is executed, as in Section V, and once again obtain timing metrics for each task and for each TDG.

The selection phase starts by first filtering the cases of TDG.json files that do not respect all the deadline constraints. If a single TDG does not respect the deadline, then that configuration (the TDG.json file) is not acceptable and so it is removed from the equation. Then, the framework will provide at the end a performance table, describing the response time (makespan) for each code version in each system configuration. Then, the selection method between all possible configurations is based on looking at the calculated makespans. The selected TDG.json file is the one providing the lowest cumulative value of makespans.

VII. Example Use Case

This section shows an example of using the described approach over a target application. To show that the approach can deal with any application besides the use of AMALTHEA and the auto-generated code, the "heat equation" benchmark is used as the target application. This benchmark is an iterative Gauss-Seidel method that calculates the heat equation.

Figure 9 shows a code parcel of an OpenMP parallel version of the heat benchmark. This method is parallelized per blocks of iterations, taking into account the dependencies between iterations. The parallelized version divides the work of the iterations between blocks. Each OpenMP task executes a block of iterations, and their execution order is controlled by the "in" and "inout" dependencies of the task.

To ease the understanding of the interdependencies between tasks, Figure 10 presents the TDG that is automatically generated by the `omp task_graph` pragma, when considering a block size of 4 by 4, for a spacial resolution of 4096*4096 points. In this graph it is possible to see that the example considers 16 working blocks (NB=4), each one with more than 1 million points to work with, to easily illustrate the TDG format. However, the experiments use NB=8, which provides for 64 blocks.

This use case has only one TDG and only one version of the code (i.e. there are no variants with tasks running in the

```
int bx,by; bx = by = NP/NB;
#pragma omp taskgraph tdg_type(static)
for (int ii=0; ii<NB; ii++) {
 for (int jj=0; jj<NB; jj++) {
  int inf_i = 1 + ii * bx; int sup_i = inf_i + bx;
  int inf_j = 1 + jj * by; int sup_j = inf_j + by;
  #pragma omp task
      depend(in: u[inf_i-bx][inf_j], u[sup_i][inf_j],
                 u[inf_i][inf_j-by], u[inf_i][sup_j])
      depend(inout: u[inf_i][inf_j])
  {
   for (int i = inf_i; i < sup_i; ++i) {
    for (int j = inf_j; j < sup_j; ++j) {
     u[i][j] = 0.25 * (u[i][j-1] + u[i][j+1]
                     + u[i-1][j] + u[i+1][j]);
    }
   }
  }
 }
}
```

Fig. 9. OpenMP version of the heat benchmark. NP*NP is the spacial resolution (number of points) and NB*NB is the number of blocks. The code is divided into NB*NB tasks, where each task has a block of bx*by iterations.

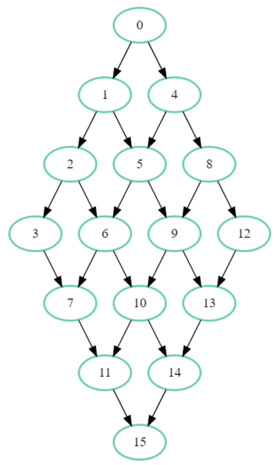

Fig. 10. The TDG for heat, when considering a block size of 4*4.

GPU), which means that it will test only different system configurations related to the CPU. The focus is on reconfiguring the frequency of the CPU and evaluate a different number of OpenMP threads.

The target platform was an NVIDIA Jetson AGX Xavier with an 8-core NVIDIA Carmel Armv8.2 64-bit CPU up to 2.5GHz and 32GB 256-bit RAM. The following were the system parameters decided for the experiments:

- frequencies: 729.6, 1190.4 and 2265.4 frequencies;
- num_threads: 4 and 8

Running the proposed approach with a configuration reflecting the specified setup will build six TDG files, where all of them have the same code version (no variants) and each one reflects executions with a specific frequency and number of threads. The result of executing the approach is a table of results as the one shown in Table II. This table is an output of the optimization flow and shows the behavior of each program version in the different environment setups.

In this case, the lowest frequency did not provide an

TABLE II
RESPONSE TIME RESULTS FOR THE HEAT BENCHMARK WHEN USING THE PROPOSED APPROACH. THE DEADLINE WAS DEFINED AS 60MS IN THE CONFIGURATION FILE.

CPU frequency (MHz)	Number of threads	response time (ms)	respects deadline
729.60	4	128.79	
1190.40	4	85.48	✓
2265.60	4	47.55	✓
729.60	8	77.97	
1190.40	8	53.31	✓
2265.60	8	26.71	✓

acceptable response time and so it is filtered out by the selection (the system will never have an acceptable execution with this low frequency). In the end, the selected version will be the TDG.json file that was executed in a CPU frequency of 2265.6MHz and using eigth OpenMP threads. Furthermore, the TDG.json is also annotated with the recommended static thread mapping.

VIII. CONCLUSIONS

This paper presented an important asset for the optimization of task-based OpenMP parallel applications, especially for applications in the context of the AMPERE project. It eases the process for the developer to explore and analyse different program variations (with tasks being able to execute either in CPU or an accelerator) and different system setup configurations in which the application might have to execute.

The framework provides a recommendation of the ideal static thread mapping for the application for each of the possible program variations and system configurations. This mapping not only aims to provide execution time efficiency but also a more predictable execution. The framework allows the use of any type of system parameters and automatically explores the combination of those parameters with all the possible variations of the application.

While the framework is already able to provide efficient and predictable static mappings, there are other constraints to be considered in the analysis, such as energy efficiency. The work in AMPERE intends to add energy efficiency as both an optimization criterion and as a constraint, therefore providing an analysis and optimization considering multiple simultaneous criteria.

REFERENCES

[1] E. Quiñones, S. Royuela, C. Scordino, P. Gai, L. M. Pinho, L. Nogueira, J. Rollo, T. Cucinotta, A. Biondi, A. Hamann, *et al.*, "The ampere project:: A model-driven development framework for highly parallel and energy-efficient computation supporting multi-criteria optimization," in *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 201–206, IEEE, 2020.

[2] C. Wolff, L. Krawczyk, R. Höttger, C. Brink, U. Lauschner, D. Fruhner, E. Kamsties, and B. Igel, "Amalthea—tailoring tools to projects in automotive software development," in *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 2, pp. 515–520, IEEE, 2015.

[3] AMPERE Consortium, "Deliverable D2.2, First release of the meta parallel programming abstraction and the single-criterion performance-aware component," 2021.

[4] C. Yu, S. Royuela, and E. Quiñones, "Taskgraph: A low contention openmp tasking framework," *arXiv preprint arXiv:2212.04771*, 2022.

[5] B. S. Center, "Bsc extended llvm 16.0." url=http://gitlab.bsc.es/ampere-sw/wp2/llvm, 2023.

[6] B. S. Center, "Task record and replay mechanism in llvm." url=https://github.com/llvm/llvm-project/commit/36d4e4c9b5f6cd0577b6029055b825caaec2dd11, 2023.

[7] B. P. Tools, "Extrae," 2019.

[8] A. Marongiu, G. Tagliavini, and E. Quiñones, "Openmp runtime," in *High Performance Embedded Computing*, pp. 145–172, River Publishers, 2022.

[9] M. S. Gharajeh, S. Royuela, L. M. Pinho, T. Carvalho, and E. Quiñones, "Heuristic-based task-to-thread mapping in multi-core processors," in *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–4, IEEE, 2022.

[10] T. Jammer, C. Iwainsky, and C. Bischof, "A comparison of the scalability of openmp implementations," in *Euro-Par 2020: Parallel Processing* (M. Malawski and K. Rzadca, eds.), (Cham), pp. 83–97, Springer International Publishing, 2020.