

Schedulability Analysis of OpenMP Applications Under Heuristic Task-To-Thread Mapping

Mohammad Samadi¹ ✉ 

Polytechnic Institute of Porto & INESC TEC, Porto, Portugal

Tiago Carvalho ✉ 

Polytechnic Institute of Porto & INESC TEC, Porto, Portugal

Luís Miguel Pinho ✉ 

Polytechnic Institute of Porto & INESC TEC, Porto, Portugal

Sara Royuela ✉ 

Barcelona Supercomputing Center, Barcelona, Spain

Abstract

Task-to-thread mapping is a key process in parallel applications to achieve the best possible performance. This process is even more challenging when it is required to meet the schedulability and timing requirements of critical systems. In these systems, mapping tasks to threads is usually carried out using static scheduling (i.e., offline mapping) to improve system schedulability, with several approaches being presented in the literature. Nevertheless, there has been little analysis on the impact that these static mapping approaches have on the schedulability of applications exploiting OpenMP, a model increasingly seen as a suitable mechanism to leverage the potential of parallel and heterogeneous processor architectures. This paper, therefore, performs a throughout evaluation of the recently presented heuristic task-to-thread mapping working with different heuristics through allocation and dispatching phases, compared with state-of-the-art, in terms of schedulability. This process is performed using a state-of-the-art schedulability analysis methodology through an integration of our simulator and an existing schedulability toolset. This evaluation allows for identifying the static heuristic mapping approaches that achieve tighter schedulability analysis than other methods in the literature.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computing methodologies → Parallel computing methodologies; Mathematics of computing → Graph theory

Keywords and phrases OpenMP, task-to-thread mapping, heuristics, response time, schedulability

Digital Object Identifier 10.4230/OASICS.NG-RES.2026.2

Supplementary Material

Software (Source Code): <https://github.com/m-samadi/heuristic-DAG-schedulability/> [2]
archived at `swh:1:dir:f1dae4e92e096975af385f3596be96775629104c`

Funding This work was supported by the KDT Joint Undertaking (JU) program under grant agreement No 101140216 (project MATISSE), co-funded by the European Union's Horizon Europe research and innovation programme and the Portuguese Fundação para a Ciência e a Tecnologia (FCT), and the HiPART project, with reference PID2023-148117NA-I00, financed by MICIU/AEI /10.13039/501100011033 and FEDER, UE.

Acknowledgements The authors thank the anonymous reviewers for their helpful comments and feedback.

¹ Mohammad Samadi is a PhD candidate at the Universitat Politècnica de Catalunya, Barcelona, Spain.



1 Introduction

Modern embedded platforms (e.g., NVIDIA Jetson series) implement complex processor architectures combining multicore systems with accelerator devices to provide high-performance capabilities to critical applications (e.g., automotive industry) [17]. Parallel programming models can be used in these systems to effectively exploit the parallel nature of applications, efficiently orchestrating parallel tasks while meeting the timing requirements of the system. One such parallel model is OpenMP. Although originally targeting shared-memory systems for the high-performance computing domain, this language has been increasingly used in real-time systems by virtue of its programmability and flexibility, combined with the quality to allow predictable execution.

From the large number of mechanisms provided in OpenMP, the tasking model is of particular interest for real-time systems because it abstracts computation in *tasks*, i.e., concurrent units of work that can express structured data parallelism and unstructured task parallelism. Furthermore, the execution of OpenMP tasks is organized as a task dependency graph (TDG), a form of direct acyclic graph (DAG), where nodes represent task instances and edges represent data dependencies between tasks.

OpenMP threads are abstractions of user-level threads that act as proxies of the actual physical processors. Accordingly, runtime implementations of the model, first, map tasks to OpenMP threads, which are in turn mapped to operating system (OS) threads (typically, pthreads), and these are mapped to actual cores. Based on this structure, there are several levels of freedom to provide OpenMP compliant implementations. To simplify the system and enhance the schedulability of OpenMP applications, this work considers that OpenMP threads are mapped one-to-one with OS threads, and these are equally mapped one-to-one to cores, and bind to these for the entire execution of an application (i.e., cannot migrate).

During the scheduling of an OpenMP task-based application, tasks are mapped to threads. This process can be carried out through a single queue, in which all threads access the same queue to insert (*allocation phase*) and pick (*dispatching phase*) tasks, or a multi-queue system, in which each thread has its own queue where to insert and pick tasks. In both cases, tasks are allocated to a queue only when their input dependencies are met, if any. Mainstream implementations of OpenMP use both methods, e.g., GCC uses a single-queue system while LLVM uses a multi-queue one, showing that the former typically introduces much more contention due to concurrent accesses to shared resources.

Schedulability is one of the main requirements in critical systems. When these systems are implemented using parallel programming models (e.g., OpenMP), the task-to-thread mapping process, whether employing global scheduling with online mapping decisions or partitioned scheduling with offline static mapping, can be efficiently adjusted to reduce worst-case response time (WCRT) and improve system schedulability. Schedulability is a property of a real-time task set that denotes whether tasks complete within their required deadlines under a scheduling algorithm, even under worst-case conditions. This process is necessary to avoid system failure. Although not strictly needed, application response time (and especially WCRT) should be reduced by selecting an appropriate scheduler to improve schedulability [4, 18, 10, 20]. In this way, the schedulability analysis of any mapping approach can be evaluated using schedulability tools based on random graphs or benchmarks.

The main issues with current evaluations of static task-to-thread mapping and their impact on OpenMP applications are that (i) they do not consider recently presented task-to-thread mappings, which may improve the schedulability of the system compared to previous methods, and (ii) analyses do not consider large and different configurations, like diverse applications

with respect to their complexity, the number of threads used, and different possible deadlines. Accordingly, in this paper, the schedulability of OpenMP applications using recent task-to-thread mapping methods [16] are evaluated based on static scheduling, and compared to the literature. Overall, the main contributions of the paper can be represented as following:

1. Applying a methodology for evaluating static task-to-thread mappings in OpenMP using an integration of our simulator and an existing schedulability analysis toolset.
2. Analyzing schedulability of the recent heuristic task-to-thread mapping, compared to other relevant mappings, under different aspects, including small and large random graphs, based on partitioned mapping with the limited preemptive policy.
3. Evaluating all the heuristic algorithms through analysis to identify their behavior in achieving acceptable schedulability.

The structure of the paper is organized as follows. Section 2 addresses a background on the main characteristics of OpenMP. Section 3 studies the related works by describing some of the existing task-to-thread mapping methods. Section 4 explains the system model and the proposed methodology for the analysis. Section 5 evaluates the schedulability of the mapping methods under simulations through different configurations. Finally, Section 6 concludes the paper.

2 Background

OpenMP is a shared-memory parallel programming framework for high-performance computing (HPC), which has been increasingly used in heterogeneous embedded platforms. From the earlier versions of the specification, it supports a *thread model* that allows boosting loop-intensive and data-parallel applications. Later versions introduced first a *task model* to create irregular and highly dynamic parallelism and then a host-centric *accelerator model* built on top of the tasking one for offloading tasks to acceleration devices. The main metrics of interest for the tasking model are simplicity of use, compatibility with existing specifications for a “legacy” execution model, and performance [17]. Given these characteristics and the suitability of OpenMP tasks to represent real-time situations, this work focuses on the tasking model.

An OpenMP task-based program starts sequentially (initial thread) until a parallel construct is found. At this point, a team of threads is created, one implicit task is created for each thread with the code contained in the associated region, and all threads start executing their corresponding tasks. A *single* construct is typically used to allow only one thread to execute the structured block associated with the single region. That region can include multiple explicit tasks defined, for example, with the *task* directive. In this structure, the implicit task executed by the single thread is the *parent* of the nested explicit tasks, which are the *child* tasks of the former. Accordingly, the explicit tasks of the single region are *siblings* among them. Data dependencies can be defined between tasks through the *depend* clause and the *in*, *out* and *inout* modifiers, which define the directionality of the dependency. Tasks are ready to execute if they do not have any input dependency or these have been already satisfied, i.e., previous tasks without output dependencies on the same data objects have already completed their execution. The data-flow relation between tasks generated by the dependencies can be represented as a TDG. Besides the point-to-point synchronizations defined by dependencies, OpenMP allows defining memory fences through the *taskwait* directive, which forces the encountering thread to wait until all its child tasks finish. In the case of the *barrier* directive, child tasks, and any task generated inside those are waited for.

Regarding the binding of tasks to threads, OpenMP tasks can be categorized into *tied*, i.e., they can only be executed by the thread that started executing them, and *untied*, i.e., once suspended, any thread can continue their execution. Tasks are *untied* by default, unless a *tied* clause is attached to the *task* directive. The main feature of an untied task is that it favors a work-conserving strategy to map tasks to threads, enabling any idle thread in the system to execute a pending task.

3 Literature review

This section presents the main features of the task-to-thread mapping methods in OpenMP used in the evaluations of this paper.

Serrano et al. [17] have addressed the predictability of OpenMP applications through the study of two schedulers: (i) a generic scheduler without using any concrete scheduling policy, and (ii) a breadth-first scheduler (BFS) [15], in which a thread that creates a task places it in the system and continues the execution of the encountering task, and work-first scheduler (WFS) [7], in which a thread that creates a task immediately executes it, both based on the first-in, first-out (FIFO) algorithm.

Melani et al. [14] have introduced two mapping policies for an OpenMP framework using a single queue for orchestrating tasks: optimal and sub-optimal allocations. The former policy uses a non-trivial integer linear programming (ILP), while the latter one uses several heuristic techniques. The optimal allocation is computationally expensive, but minimizes the makespan of tasks and also provides a reference point to evaluate other static and dynamic algorithms. In contrast, the sub-optimal allocation is not computationally expensive and, therefore, is more suitable for large task sets. In the sub-optimal policy, a ready task-part, i.e., the sequential task region between two TSPs, is selected from the queue using one of the heuristic algorithms, and the first idle thread is selected to execute the task-part. Note that TSPs are task scheduling points that dynamically divide task regions into multiple parts, where each part can be executed with an idle thread based on the order in the same task region. Among the heuristics provided, we have used the longest processing time (LPT), which selects the task with the longest worst-case execution time (WCET), the shortest processing time (SPT), which selects the task with the shortest WCET, and the largest number of successors in the next level (LNSNL), which selects the task with the largest number of immediate successors.

In a recent work [16], we have proposed a heuristic task-to-thread mapping, an extended version of the method presented in [9], to perform the mapping process in a multi-queue OpenMP system using different heuristic algorithms. This technique consists of two phases, allocation and dispatching. During allocation, one of the allocation queues is selected for each OpenMP task using one of the following heuristics: minimum number of task-parts (MNTP), next thread (NT), most recent idle time (MRIT), minimum total execution time (MTET), maximum total response time (MTRT), and total multi-criteria decision (TMCD). During dispatching, when a thread is idle, it selects one of the ready tasks from its queue, if any, using one of the heuristics: minimum execution time (MET), maximum response time (MRT), and multi-criteria decision (MCD). Task stealing is disabled in this technique to enhance the predictability of the system. This paper focuses on the schedulability analysis of OpenMP applications using these proposed heuristics, in comparison with that obtained using the other mapping approaches.

4 Methodology

This section describes the system model used to generate random graphs (DAGs) and the tools used to evaluate the schedulability of the generated graphs.

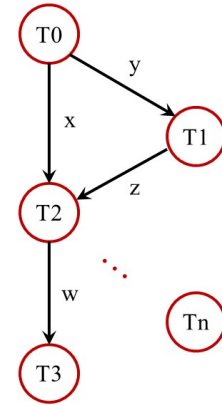
4.1 System model

A DAG is considered, where each node in the DAG is an *OpenMP task* (or briefly a *task*). It can include internal synchronization, where data dependencies exist between tasks. Therefore, tasks must execute only after their input dependencies are met. The DAG has an *end-to-end response time*, called *response time* or *application response time*, and includes a *deadline*, denoting a time instant in which the DAG must be completed, to guarantee schedulability. The schedulability of the tasking system is specified based on *implicit* and *constrained* deadlines, with a deadline equal to the period in implicit deadlines, and smaller than or equal to the period in constrained deadlines. In addition, the *homogeneous* architecture is utilized for the mapping process, in which all OpenMP tasks are executed by CPUs. The *task-centric* tasking model of OpenMP is considered in the system model, where tasks represent various parts of the application (i.e., DAG). Tasks can be executed in parallel using multiple threads, where $\{4, 8\}$ threads are used in evaluations.

This model is commonly used in OpenMP task-based programs (OpenMP-DAG), i.e., a program in which a task (*Tmain*, the implicit task generated in the parallel construct that enters the single region) is parent of a series of sibling tasks (*T0* to *Tn*), created through the *task* directive, with no nested tasks inside. Figure 1 illustrates an example for this model, in which some of the tasks expose data dependencies, e.g., *T2* has two input dependencies, so it can execute only if *T0* and *T1* have completed. Note that n is the number of tasks and m is the number of threads.

```
#pragma omp parallel num_threads (m)
#pragma omp single // Tmain
{
    #pragma omp task depend (out: x,y)
    T0();
    #pragma omp task depend (in: y; out: z)
    T1();
    #pragma omp task depend (in: x,z; out: w)
    T2();
    #pragma omp task depend (in: w)
    T3();
    ...
    #pragma omp task
    Tn();
}
```

(a) Source code.



(b) OpenMP-DAG.

■ **Figure 1** Example of the type of OpenMP system considered in this work.

4.2 Tools

To evaluate schedulability of OpenMP applications, two types of random graphs are used: (i) regular graphs, with a number of tasks in the range of $[1, 50]$ and exposing limited parallelism, and (ii) wide graphs, with a number of tasks in the range of $[51, 100]$ and exposing extensive parallelism. A DAG-scheduling library [1] is integrated in the developed

simulator [2] to generate the DAGs using the method presented by Melani et al. [13]. The following modifications are made to the default parameters: (i) the maximum recursion depth is set to 200, (ii) the maximum allowed parallel branches is set to 5 for regular graphs and 8 for wide graphs, and (iii) the schedulability test varying the total utilization of task sets is used to generate DAGs with constrained deadlines in a periodic system. The DAGs are generated based on the implicit and constrained deadlines in a periodic system [12]. Consequently, results for each method are generated over 200 iterations (i.e., runs). A new graph is generated in each iteration to perform the analysis under different conditions across all the iterations. The overall schedulability of the graphs across these iterations for each mapping method is calculated using Eq. 2, and the number of iterations is chosen based on an assumption, depending on the platform's ability to complete the executions.

As many practical parallel workloads (e.g., DAG-based perception pipelines, control loops, or GPU kernels) can be represented as a single recurring DAG, the system model focuses primarily on systems with only one graph of parallel computation, executed under a mapping algorithm. The problem of mapping one DAG with precedence constraints is already NP-hard and addresses the core challenges of real-time high-performance computation. Nevertheless, the same approach can also be used for multiple graphs of parallel computation (i.e., various DAGs).

The analysis is conducted using the limited preemptive policy based on partitioned scheduling, as presented by Casini et al. [6]. Partitioned scheduling, rather than global scheduling, is used in the evaluations for the following reasons: (i) global mapping can increase overheads due to task migration between threads and, consequently, incurring more cache misses depending on the memory and cache architecture as well as the task structure [11]; (ii) in an average case, the schedulability of partitioned scheduling for a small number of heavy tasks is higher than that of global scheduling [5]; and (iii) partitioned scheduling is the most appropriate for hard real-time systems, as these systems should be more predictable and included with a low overhead [3], while global scheduling is the most suitable for soft real-time systems, as these systems may miss the deadline in some situations [8]. The DAG-scheduling library used to generate random DAGs is also applied for schedulability analysis and response-time calculation. This library includes the implementations of several schedulability methods [19].

The method to perform the analysis is the following: First, the simulator sends a request to the library to generate a random graph, which is generated based on the defined parameters and saved; then, the simulator reads the graph information from the file and performs the mapping using the selected algorithm storing the static scheduling of tasks in threads in a YAML file; next, the simulator sends a request to the library to perform schedulability analysis and determine response time on the submitted task set (created by random graph generation), which responds by reading the graph information from the YAML file, conducting the analysis, and storing the schedulability results and the response time in two text files.

5 Performance evaluation

This section describes the schedulability analysis results for the mapping approaches under different configurations obtained through simulation-based offline mapping. Specifically, it evaluates the efficiency of the heuristic task-to-thread mapping methods [16], compared to BFS [15], LPT [14], SPT [14], LNSNL [14], and WFPA [1, 19] (i.e., an algorithm to allocate tasks to threads based on having the least used utilization). The goal is to analyze the impact of the different mappings on the schedulability and WCRT of random graphs; therefore, an

existing schedulability analysis tool [1] is used to compare the different algorithms under the same conditions. Note that the results for the NT-X heuristics in the heuristic mapping are not included here, as they do not perform as efficiently as the other algorithms in most cases.

5.1 Setup

The evaluation process is performed based on schedulability analysis for each mapping method, where the comparison between the WCRT obtained with each method and that obtained with the WFPA algorithm is used in the evaluations. WCRT is equal to the maximum response time obtained by the mapping approach across various iterations. WFPA is considered as a baseline in evaluations as it is the algorithm originally used in the schedulability analysis tool and survey [1, 19]. Note that the analysis is conducted based on utilization of 1, 2, 4, and 8. The utilization of the graphs is calculated as follows [19]:

$$Utilization = \frac{Vol}{T} \quad (1)$$

where Vol is the volume and T is the period of the graph. The volume is determined based on the sum of the execution times of the tasks in the graph [19]. The schedulability given with each method is determined as follows:

$$Schedulability = \frac{\sum_{i=1}^{itr} SchedVal_i}{itr} * 100 \quad (2)$$

where itr indicates the number of iterations and $SchedVal_i$ indicates the schedulability of the graph in iteration i (the schedulability is equal to 1 if the graph is schedulable; otherwise, it is equal to 0). Note that the graph is schedulable if the execution completes within the relative deadline. Furthermore, the difference in WCRT is determined as follows:

$$Diff = 1 - \frac{WCRT_{WFPA}}{WCRT_{meth}} * 100 \quad (3)$$

where $WCRT_{WFPA}$ is the WCRT obtained with WFPA and $WCRT_{meth}$ is the WCRT obtained with each method. Table 1 shows the characterization of the DAGs used for the schedulability analysis.

■ **Table 1** The cases considered in the schedulability analysis.

Case #	Size of graph	Number of tasks	Deadline policy
1	Small	1–50	Implicit
2	Large	51–100	Implicit
3	Small	1–50	Constrained
4	Large	51–100	Constrained

The evaluations are performed on a 12th Gen Intel(R) Core(TM) i7-12650H processor running at 2.30 GHz. Since the library uses only mathematical analysis and does not influence memory accesses, it does not matter which hardware platform is used. Consequently, the evaluations are built on a platform that was more accessible for experimentation. However, the analysis is also performed based on different number of threads (i.e., 4 and 8 threads).

5.2 Results

This subsection evaluates the results for single DAGs obtained with the mapping approaches in terms of schedulability analysis and WCRT. As the schedulability achieved for multiple DAGs is very low, only single DAGs are discussed in this paper.

Figure 2 shows the schedulability results obtained using various mapping methods in different cases, where the utilization of the task set is considered 1 (the volume of the graph equals its period). Since the graphs are not schedulable with any of the mapping methods when the utilization is $\{2, 4, 8\}$, their results are not included here. The reason is that, in these cases, the volume of the graphs is much larger than their period and, therefore, their execution is not completed before the relative deadline.

Since schedulability analysis is performed using a tool, and not based on actual execution on a hardware platform, the schedulability primarily depends on the DAG specifications and the OpenMP tasking model. However, the number of threads is considered in the evaluations and therefore can also be considered a hardware specification.

	Case 1 with 4 threads	Case 2 with 4 threads	Case 2 with 8 threads	Case 3 with 4 threads	Case 4 with 4 threads	Case 4 with 8 threads
BFS	100%	93%	100%	35%	4%	10%
LPT	100%	91%	100%	34%	2%	17%
SPT	100%	88%	100%	38%	2%	15%
LNSNL	100%	91%	100%	37%	5%	15%
MNTP-MET	100%	99%	100%	27%	3%	24%
MNTP-MRT	100%	100%	100%	28%	4%	25%
MNTP-MCD	100%	100%	100%	28%	4%	25%
MTET-MET	100%	100%	100%	32%	6%	25%
MTET-MRT	100%	100%	100%	33%	5%	25%
MTET-MCD	100%	100%	100%	33%	5%	25%
TMCD-MET	100%	99%	100%	32%	6%	25%
TMCD-MRT	100%	100%	100%	34%	2%	24%
TMCD-MCD	100%	100%	100%	34%	2%	24%
WFPFA	100%	87%	100%	37%	1%	19%

■ **Figure 2** Schedulability analysis results obtained with the mapping approaches in different configurations.

Figure 3 also illustrates the evaluation results related to the difference in WCRTs obtained using the mapping approaches. In the comparison results, (i) negative difference (started with “-”) means that the WCRT obtained with a mapping method is reduced compared to the baseline (i.e., WFPFA), (ii) positive difference means that the WCRT achieved using the mapping method is increased compared to the baseline, and (iii) “0” means that there is no difference between the performance of the mapping method and the baseline. The comparison results indicate that the WCRT of graphs obtained using the proposed heuristic mapping is lower than that obtained with the other methods, especially in cases 2 and 4 with 4 threads.

The comparison results achieved in Figures 2 and 3 indicate that the schedulability of graphs achieved through the heuristics proposed in [16] is higher than that achieved through the other methods in most cases. The schedulability results for Case 1 (i.e., small DAGs with implicit deadlines) when scheduled with 4 threads reveal no significant difference in mapping performance. However, the WCRT results indicate that the WCRT obtained with these heuristics is lower than that reported in other works in the literature in most cases, achieving high quality with the heuristic mapping. Note that MTET-X and TMCD-X perform better than MNTP-X, indicating that some metrics, such as task execution time, can efficiently reduce the WCRT.

The results for Case 2 (i.e., large graphs with implicit deadlines) with 4 threads show that all heuristics achieve higher efficiency than the other methods, as most graphs are schedulable with the heuristics when the utilization is set to 1. Furthermore, the WCRT given with these

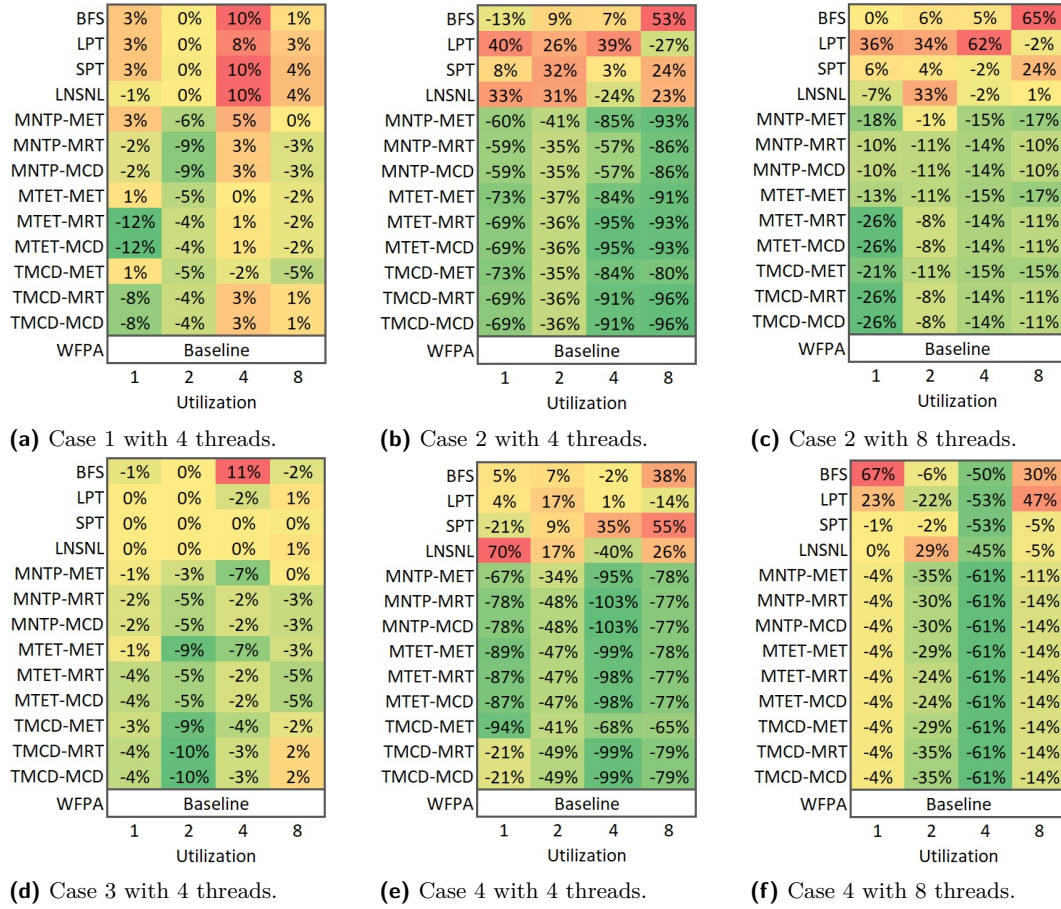


Figure 3 Difference in WCRT obtained using the mapping methods. The WCRT difference is the difference between the WCRT obtained with each method and that obtained with the WFFPA algorithm.

heuristics is lower than that obtained with the other methods. A comparison between these results and the previous results for small DAGs shows that the efficiency of the heuristics in large DAGs is generally better than that in small DAGs. The reason is that the dispatching phase in the heuristic mapping is better suited to cases where queues contain many tasks and there are multiple options for selecting them. The use of specific timing metrics (e.g., task execution time) has made these heuristics more suitable for most graphs, including those involving both light and heavy tasks.

The results for Case 2 (i.e., large graphs with implicit deadlines) scheduled with 8 threads show that the schedulability obtained using all methods is the same across different utilization. These results, compared to the same graphs but with 4 threads, demonstrate that the schedulability rate improves as the number of threads increases. The reason is that the heuristics can better exploit the capabilities of the possible mappings. Furthermore, the difference between the WCRT obtained with these heuristics and that obtained with the preliminary methods is noticeable. Therefore, the results indicate that the heuristics perform more efficiently at minimizing WCRT, particularly when large DAGs and a high number of threads (i.e., 8) are used. The reason is that these situations provide more freedom to select the most suitable allocation queues for ready tasks, as well as the most appropriate ready tasks from the queues.

The schedulability of the heuristics is slightly lower than that of some previous methods when Case 3 (i.e., small graphs with constrained deadlines) is scheduled with 4 threads. However, MTET-X and TMCD-X provide very similar results. In contrast, all these heuristics perform more efficiently in minimizing WCRT than the other methods, as shown in the previous results. The results for this configuration and for small graphs with implicit deadlines indicate that the heuristics from the heuristic mapping perform more efficiently under the implicit deadline policy. The reason is that small DAGs include only a few tasks, so these heuristics cannot select the most appropriate queues for ready tasks or suitable ready tasks within those queues. Therefore, the system misses its deadline under the constrained deadline policy.

The schedulability obtained with all methods for Case 4 (i.e., large DAGs with constrained deadlines) is low when running with 4 threads. The performance of the heuristics, especially MTET-X, is slightly better than some other methods, including LPT, SPT, and WFP. The main reason is that these heuristics perform efficiently in large DAGs, so a shorter deadline, as in the constrained deadline system, has little effect on their efficiency. However, the heuristics perform more efficiently in the graphs with implicit deadlines. Moreover, the WCRT obtained with these heuristics is lower than that obtained with the other methods, especially at a graph utilization of 4.

The schedulability of the graphs obtained with the heuristics proposed in [16], compared to that obtained with the other methods, especially BFS, SPT, and LNSNL, for Case 4 (i.e., large DAGs with constrained deadlines), running with 8 threads, is noticeable. In addition, the difference in the WCRT achieved with these heuristics and the previous ones is very considerable. Comparing this case with the earlier case that ran with 4 threads shows that the heuristics perform more efficiently with 8 threads. The reason is that large DAGs, including many tasks, allow for a more suitable allocation in the ready queues when the number of threads is high.

Furthermore, the performance of all the mapping algorithms for Case 4 executed with 4 threads is worse than the performance with 8 threads. The reason is that DAGs are completed faster using a large number of threads (8 threads) than using a smaller number of threads (4 threads) and are, therefore, more schedulable, meeting the demanded deadline.

Overall, the analysis results show that (i) the schedulability obtained with most of the heuristics from the heuristic task-to-thread mapping for large DAGs with implicit deadlines running using 4 threads is higher than that obtained with the preliminary methods, (ii) the schedulability given with most of these heuristics for large DAGs with constrained deadlines, especially in the graphs running with 8 threads, is higher than that given with the previous methods, (iii) most of the graphs generated based on the utilization of 2, 4 and 8 are not schedulable with any of the methods, (iv) the WCRT is minimized with most of these heuristic methods, compared to WFP and the other non-heuristic methods, especially in the cases for large DAGs with 4 threads, and (v) the schedulability of DAGs generated based on implicit deadlines is higher than that generated based on constrained deadlines, in all the cases.

6 Conclusion

This paper evaluates the recent heuristic task-to-thread mapping method, against other mapping methods, in terms of schedulability and WCRT. The evaluation is conducted based on various types of DAGs (including small and large graphs) under both implicit and constrained deadline policies. The results for single DAGs show that the efficiency of the heuristic mapping, in terms of improving schedulability and minimizing WCRT, are better than those of other methods in most cases.

In particular, the schedulability analysis results show that, when the utilization of task sets is 1, the efficiency of the proposed heuristics is better than the previous mapping methods in Case 2 (i.e., large graphs with implicit deadline) with 4 threads and Case 4 (i.e., large graphs with constrained deadline) with 8 threads. The reason is that when there are many tasks in the system (i.e., large graphs), the heuristic algorithms can select the most appropriate tasks from the queues. Therefore, the mapping's work-conserving nature is improved, the program completes sooner (i.e., WCRT is reduced), and schedulability is improved.

Furthermore, these heuristics are more efficient than or equal to the other methods in Case 4 with 4 threads. The reason is that when the number of threads is low, the heuristic algorithms cannot select the most suitable queues for tasks, leading to a slight increase in WCRT and a reduction in schedulability, compared to the case with many threads. Their efficiency is comparable to that of the previous methods in Case 1 (i.e., small graphs with implicit deadlines) with 4 threads and Case 2 with 8 threads. The reason is that when the number of tasks is low (i.e., small graphs), the heuristics fail to select the most proper tasks during the allocation and dispatching phases.

Besides, the WCRT obtained using the heuristic algorithms is lower than that of previous mapping methods in most cases, especially in cases 2 and 4. The reason is that the most appropriate tasks can be selected using these algorithms, especially during the dispatching phase, based on their characteristics (e.g., execution time), when there are many tasks in the system. This flexibility improves the work-conserving nature of the mapping and considerably reduces the WCRT. All these evaluations indicate that the schedulability and quality of critical systems can be improved by utilizing the heuristic task-to-thread mapping.

As future work, schedulability analysis of real-world programs, especially those used in artificial intelligence applications with enormous workloads, will be conducted under different application and system configurations. Furthermore, schedulability analysis will be performed based on other policies, such as global or semi-partitioned approaches.

References

- 1 The dag-scheduling library, 2025. URL: <https://github.com/mive93/DAG-scheduling/>.
- 2 The heuristic-dag-schedulability simulator, 2025. URL: <https://github.com/m-samadi/heuristic-DAG-schedulability/>.
- 3 Luca Abeni and Tommaso Cucinotta. Edf scheduling of real-time tasks on multiple cores: Adaptive partitioning vs. global scheduling. *ACM SIGAPP Applied Computing Review*, 20(2):5–18, 2020. doi:10.1145/3412816.3412817.
- 4 Neil Audsley, Alan Burns, Rob Davis, Ken Tindell, and Andy Wellings. Real-time system scheduling. In *Predictably dependable computing systems*, pages 41–52. Springer, 1995. doi:10.1007/978-3-642-79789-7_3.
- 5 Artem Burmyakov and Borislav Nikolić. An exact comparison of global, partitioned, and semi-partitioned fixed-priority real-time multiprocessor schedulers. *Journal of Systems Architecture*, 121:102313, 2021. doi:10.1016/j.sysarc.2021.102313.
- 6 Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. Memory feasibility analysis of parallel tasks running on scratchpad-based architectures. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 312–324. IEEE, 2018. doi:10.1109/RTSS.2018.00047.
- 7 Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation (PLDI '98)*, pages 212–223, 1998. doi:10.1145/277650.277725.
- 8 Mohammad Samadi Gharajeh, Tiago Carvalho, and Luis Miguel Pinho. Configuration of parallel real-time applications on multi-core processors. In *2022 IEEE 20th International Conference on Industrial Informatics (INDIN)*, pages 67–73. IEEE, 2022. doi:10.1109/INDIN51773.2022.9976163.

- 9 Mohammad Samadi Gharajeh, Sara Royuela, Luis Miguel Pinho, Tiago Carvalho, and Eduardo Quiñones. Heuristic-based task-to-thread mapping in multi-core processors. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4. IEEE, 2022. doi:10.1109/ETFA52439.2022.9921453.
- 10 Tei-Wei Kuo, Li-Pin Chang, Yu-Hua Liu, and Kwei-Jay Lin. Efficient online schedulability tests for real-time systems. *IEEE Transactions on Software Engineering*, 29(8):734–751, 2003. doi:10.1109/TSE.2003.1223647.
- 11 Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 85–96. IEEE, 2014. doi:10.1109/ECRTS.2014.23.
- 12 Cláudio Maia, Patrick Meumeu Yoms, Luís Nogueira, and Luis Miguel Pinho. Semi-partitioned scheduling of fork-join tasks using work-stealing. In *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, pages 25–34. IEEE, 2015. doi:10.1109/EUC.2015.30.
- 13 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C Buttazzo. Response-time analysis of conditional dag tasks in multiprocessor systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 211–221. IEEE, 2015. doi:10.1109/ECRTS.2015.26.
- 14 Alessandra Melani, Maria A Serrano, Marko Bertogna, Isabella Cerutti, Eduardo Quinones, and Giorgio Buttazzo. A static scheduling approach to enable safety-critical openmp applications. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 659–665. IEEE, 2017. doi:10.1109/ASPDAC.2017.7858399.
- 15 Girija J Narlikar. Scheduling threads for low space requirement and good locality. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures (SPAA '99)*, pages 83–95, 1999. doi:10.1145/305619.305629.
- 16 Mohammad Samadi, Sara Royuela, Luis Miguel Pinho, Tiago Carvalho, and Eduardo Quiñones. Time-predictable task-to-thread mapping in multi-core processors. *Journal of Systems Architecture*, 148:103068, 2024. doi:10.1016/j.sysarc.2024.103068.
- 17 Maria A Serrano, Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna, and Eduardo Quinones. Timing characterization of openmp4 tasking model. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 157–166. IEEE, 2015. doi:10.1109/CASES.2015.7324556.
- 18 Danbing Seto, John P Lehoczky, Lui Sha, and Kang G Shin. On task schedulability in real-time control systems. In *17th IEEE real-time systems symposium*, pages 13–21. IEEE, 1996. doi:10.1109/REAL.1996.563693.
- 19 Micaela Verucchi, Ignacio Sañudo Olmedo, and Marko Bertogna. A survey on real-time dag scheduling, revisiting the global-partitioned infinity war. *Real-Time Systems*, 59(3):479–530, 2023. doi:10.1007/s11241-023-09403-3.
- 20 Fengxiang Zhang and Alan Burns. Schedulability analysis for real-time systems with edf scheduling. *IEEE Transactions on Computers*, 58(9):1250–1258, 2009. doi:10.1109/TC.2009.58.