2021 IEEE Real-Time Systems Symposium (RTSS) | 978-1-6654-2802-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/RTSS52674.2021.00022

Calculating Worst-Case Response Time Bounds for OpenMP Programs with Loop Structures

Jinghao Sun[1], Nan Guan[2†], Zhishan Guo[3], Yekai Xue[4], Jing He[1], Guozhen Tan[1]

[1]Dalian University of Technology, China; [2]City University of Hong Kong, Hong Kong;
[3]University of Central Florida, U.S; [4]Northeastern University, China

***Abstract*—OpenMP is a promising framework for developing parallel real-time software on multi-cores. Recently, many graph-based task models representing realistic features of OpenMP task systems have been proposed and analyzed. However, all previous studies did not model the `loop` structures, which is common in OpenMP task systems. In this paper, we formulate the workload of OpenMP task systems with `loop` structures as the cyclic graph model and study how to compute safe upper bounds for the worst-case response time (WCRT). The loop structures combined with the creation of tasks and conditional branches result in a large state space. Simply unrolling the `loop` and/or enumerating all the possible execution flows would be computationally intractable. As the major technical contribution, we develop a linear-time dynamic programming algorithm to compute the WCRT bound without unrolling `loops` or explicitly enumerating the execution flows. Experiments with both synthetic task graphs and realistic OpenMP programs are conducted to evaluate the performance of our method.**

***Index Terms*—OpenMP, response time bound, loop structure**

I. Introduction

Multi-cores are becoming mainstream hardware platforms for embedded and real-time systems. To fully utilize the processing capacity of multi-cores, software should be parallelized. OpenMP [1], the de facto standard of parallel programming frameworks on shared memory architectures, appears to be promising for developing efficient parallel embedded and real-time software on multi-cores. OpenMP supports explicit parallel task systems since version 3.0. Directed Acyclic Graphs (DAG) is a widely used workload model to represent parallel tasks. This motivates much theoretical work on real-time scheduling and analysis of DAG task models [2]–[17].

However, the characteristics of OpenMP task systems cannot be fully captured by the standard DAG task model. Unlike traditional real-time system models in which the program-level structures (e.g., `if-else` and `loops`) and system-level behaviors (e.g., the activation of tasks and multitasking execution) are handled separately, the workload generated by OpenMP task systems is tightly coupled with the program-level structure. Fig. 1 shows an example OpenMP program, where three tasks τ_1, τ_2, and τ_3 are created from different branches of an `if-else` structure wrapped in a `loop`. Previous work has studied OpenMP real-time task systems with `if-else` [18]–[23], but it is still an open problem to model and analyze OpenMP task systems in which creation of tasks is nested in `loops` as exampled in Fig. 1. Realistic OpenMP programs often have creation of tasks nested in `loops`. For example, in the BOTS benchmark [24], 10 out of 12 programs have such structures.

```
for(i=0;i<100;i++){
    if(cond-exp)
        #pragma omp task{code1;} //τ1
    else
        #pragma omp task{code2;} //τ2
        #pragma omp task{code3;} //τ3
}
```

Fig. 1: An example OpenMP program with `loop` structure.

In this paper, we study how to bound the worst-case response time (WCRT) of OpenMP task systems with `loops`. Our work is built upon the classical response time bound by Graham for the DAG task model [25]. Applying Graham's bound to OpenMP task systems with `loops` is a challenging problem. The workload abstraction of OpenMP task systems is *cyclic* graphs (due to `loops`), instead of the acyclic graphs considered in previous work [23]. On the one hand, it is not wise to convert cyclic graphs into acyclic graphs (by unrolling `loops`) and then directly reuse the existing acyclic-graph-based analysis method (e.g., in [23]). The reason is that simply unrolling `loops` will result in very large acyclic graphs (especially in the presence of nested `loops`) and make the analysis problem computationally intractable. On the other hand, there is no evidence that the WCRT can be estimated without exploring the internal structure of `loops`. Especially, when a `loop` contains `if-else`, the (critical parameters used in) WCRT bound is not necessarily derived by taking the same branch in different iterations of the `loop` (We will illustrate this in Sec. V). The combination of `if-else` and `loops` results in a very large state space of the possible run-time behaviors, e.g., if we wrap the `if-else` structure in the `loop` with iteration number 100 as shown in Fig. 1, there are 2^{100} possible execution flows depending on which branch is taken in each iteration of the `loop`. It is highly intractable to explicitly enumerate them to derive the WCRT bound.

To address the above challenges, we propose efficient methods to compute the desired Graham's WCRT bound of the OpenMP tasks with `loops`. The main contribution of this paper is to design a linear-time dynamic programming

† Corresponding author: Nan Guan, nanguan@cityu.edu.hk

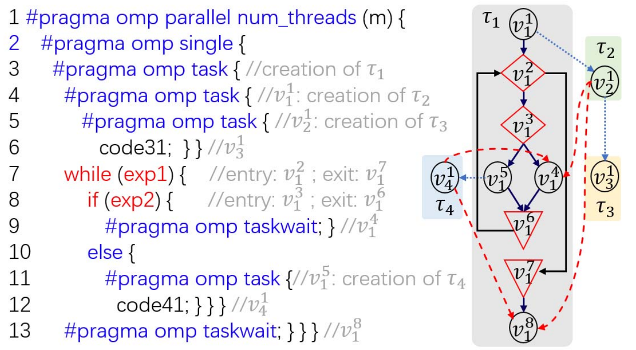
algorithm for more exactly calculating the parameters (e.g., the volume and the length) used in the WCRT bound. Our method does not need to unroll `loops` and thus the timing complexity is independent from the loop bounds. This is achieved by exploring deep insights about the workload characteristics of the OpenMP task systems and designing proper abstractions to efficiently explore the control-flow of the program. Experiment results show the effectiveness of our analysis techniques, i.e., comparing with the baseline method that can only derive parameters' upper bound, our method (for computing parameters more exactly) can always significantly improve the WCRT bound.

II. Related Work

Vargas et. al [12] for the first time modeled OpenMP task systems by DAGs. Serrano et. al [11] developed WCRT bound analysis for the OpenMP DAG task model. They studied the OpenMP tasks under Breadth-First-Scheduling (BFS) and Work-First-Scheduling (WFS), and pointed out that when there are only `untied` tasks, the WCRT of an OpenMP system can be well bounded by the classical Graham's bound for traditional DAG model [25]. However, when there are `tied` tasks, they showed that the OpenMP task systems have unacceptably large WCRT. Sun et. al [13] proposed BFS* algorithm, and derived a WCRT bound for `tied` tasks. All above work assumes that OpenMP programs do not have `if-else` clauses. The task models with both intra-task parallelism and `if-else` structures are studied in [19]–[22]. These models all assume "well composed" graph structures recursively composed by single-source-single-sink parallel and conditional components. Sun et. al [23] proposed a dynamic programming to deal with the non-well-composed DAGs, but they cannot handle `loop` structures. In this paper, we can analyze the non-well-composed DAGs that have not only `if-else` structures, but also `loop` structures. Vargas et.al [26] also studied the OpenMP program with `if-else` and `loop` structures. This paper works on a different level from Vargas's work [26], which aims to reduce the memory consumption in the OpenMP runtime, and does not focus on the WCRT analysis. Moreover, the DAG model in [26] does not consider the inner-task control flow structure, i.e., each task is formulated as a vertex. Instead, in this paper we model each task as a control flow graph, and reveal tasks' inner structure. [27] studies OpenMP programs with nested parallel regions and the `parallel loop` structure, which belongs to a work-sharing structure, meaning that the iterations of a `loop` are executed in parallel. It is totally different from the `loop` (which is a control flow structure) discussed in this paper.

III. Overview of OpenMP Systems

In this paper, we focus on OpenMP programs using `parallel`, `task` and `taskwait` directives[1], which are supported by OpenMP 3.0 [1] and above. Fig. 2 gives an example OpenMP program, where the `parallel` directive (Line 1) constructs the associated `parallel` region from Lines 2 to 13. The `parallel` region is further partitioned into a set of parallel units, called *tasks*. Each task corresponds to the *code region* immediately enclosed in the brackets following a `task` directive. We use $\mathcal{T}$ to denote the set of OpenMP tasks, i.e., $\mathcal{T} = \{\tau_1, \cdots, \tau_n\}$, where τ_k is the k-th task in $\mathcal{T}$, and $n = |\mathcal{T}|$ is the task number. OpenMP supports nested tasks. More precisely, task τ_l is *nested* in task τ_k if the `task` directive of τ_l is contained in τ_k's code region. In this case, we say τ_k is the parent of τ_l, and τ_l is the *child* of τ_k. We assume that a task has at most one parent. The task with no parent is called the *main* task of $\mathcal{T}$. For example, Fig. 2 shows a task set with 4 tasks. The main task τ_1 has two children τ_2 and τ_4. τ_2 is the parent of τ_3. The code region of τ_1 includes Lines 4, 7, 8, 9, 10, 11 and 13.



```
1 #pragma omp parallel num_threads (m) {
2   #pragma omp single {
3     #pragma omp task { //creation of τ1
4       #pragma omp task { //v_1^1: creation of τ2
5         #pragma omp task { //v_2^1: creation of τ3
6           code31;  } } //v_3^1
7       while (exp1) {    //entry: v_1^2 ; exit: v_1^7
8         if (exp2) {      //entry: v_1^3 ; exit: v_1^6
9           #pragma omp taskwait; } //v_1^4
10        else {
11          #pragma omp task {//v_1^5: creation of τ4
12            code41; } } } //v_4^1
13      #pragma omp taskwait; } } } //v_1^8
```

Fig. 2: An example OpenMP program and its digraph model.

Control Flow Structure. The code region of a task (also called the task region) can be seen as a *block* with a single entry and a single exit, which correspond to the first statement and the last statement of the task region, respectively. Moreover, a task region may contain conditional structures, e.g., `if-else` and `loop` structures, and both of them can also be seen as single-entry-single-exit blocks. For the sake of simplicity, we assume that an `if-else` block exactly has two branches. A `loop` block exactly has a *body* with the loop bound K, i.e., the loop body should not be executed more than K times. Conditional blocks are allowed to be nested. For example, Fig. 2 shows an `if-else` block (Line 8) nested in a `loop` block (Line 7). We assume that the conditions of all conditional blocks are independent, e.g., the value of the `loop` condition `exp1` does not affect the value of the `if` condition `exp2`, and vise versa.

A conditional block is *irregular* if it contains OpenMP directives (e.g., `task` and `taskwait`). We say task system $\mathcal{T}$ is *well composed* if there is no irregular block in $\mathcal{T}$. Otherwise, $\mathcal{T}$ is *non-well composed*. For example, Fig. 2 shows an irregular `if-else` block, which contains both `task` and `taskwait` directives. Consequently, the task system in Fig. 2 is non-well composed. This paper tackles both well composed and non-well composed systems.

[1]The syntax of an OpenMP directive in C language is "`#pragma omp` directive-name". Instead of the full syntax of a directive, we only use the "directive-name" for simplicity. Moreover, additional clauses including `target`, `taskyield`, `taskgroup`, `critical` and `depend` are out of the scope of this paper.

Execution Semantics. The execution of OpenMP task system $\mathcal{T}$ begins with the execution of its main task τ_1. Other tasks are executed only if its `task` directive is executed. For any task τ_k of $\mathcal{T}$, the execution of τ_k means to execute τ_k's code region from the entry to the exit. When encountering an `if-else` block, exactly one of its branches is executed. When encountering a `loop` block with loop bound K, its body is executed at most K times. Moreover, when encountering a `task` directive, a child task of τ_k is created, which must be executed later. When encountering a `taskwait` directive, the task τ_k is suspended until all τ_k's children that are created beforehand have been finished.

Runtime. At OpenMP runtime, tasks created from the OpenMP task system are scheduled to execute on threads, which are execution entities (mapped to, e.g., a thread in the underlying OS) to execute OpenMP tasks. Similar to previous work [11], [12], we assume each thread to exclusively execute on a dedicated core. In OpenMP, a task is either `tied` or `untied`. The `tied` task forces its code to be executed on the same thread. In contrast, an `untied` task can migrate among threads. There are two types of OpenMP-compliant scheduling algorithms. One is called WFS [28], which prefers to execute newly created tasks. The other one is BFS [29], which tends to execute tasks that have been executed on the threads. The common feature of WFS and BFS is that they are both work-conserving when scheduling `untied` tasks, i.e., no thread will be left idle if there are tasks eligible for execution. For `tied` tasks, BFS and WFS not only lose the work-conserving property, but also may lead to extremely bad timing behaviors (in the worst-case all parallel workloads are executed on the same thread) [11]. We only consider `untied` tasks in the rest of this paper.

IV. System Model

We formulate the OpenMP task system $\mathcal{T}$ as a digraph such that each task τ_k of $\mathcal{T}$ has a control flow graph (CFG) structure, and there are inter-dependencies among tasks of $\mathcal{T}$. In the following, we first introduce the intra-task structure of a task τ_k, and then introduce the inter-task structure of the whole task system $\mathcal{T}$.

Intra-task structure. The CFG of a task τ_k is formulated as a tuple $\tau_k = (V_k, E_k)$, where V_k is the set of vertices, and E_k is the set of control flow (CF) edges. Here we use the same symbol τ_k to denote the CFG of task τ_k for reducing notations. Each vertex v_k^i of V_k represents a sequential code segment of task τ_k, and has the worst-case execution time c_k^i. Each CF edge $(v_k^i, v_k^j) \in E$, denoted by a solid-line arrow in Fig. 2, represents the dependency between vertices v_k^i and v_k^j. We say v_k^i is the *predecessor* of v_k^j if there is an edge from v_k^i to (the predecessor of) v_k^j, and in this case, v_k^j is called the *successor* of v_k^i. A vertex is called the *source* vertex of τ_k, denoted as v_k^{src}, if it has no predecessors (via CF edges). A vertex is called the *sink* vertex of τ_k, denoted as v_k^{snk}, if it has no successors (via CF edges). There is a single source vertex and a single sink vertex in τ_k, in the sense that the code region of τ_k is a block with a single entry and a single exit (as described in Sec. III). Similarly, for each subgraph B of τ_k, we call a vertex v_k^i of B as the *entry* vertex of B if v_k^i's predecessors are all outside B, and we call v_k^i as the *exit* vertex of B if v_k^i's successors are all outside B.

We distinguish two types of vertices of τ_k: *conditional* vertices and *non-conditional* vertices. The non-conditional vertex is represented by a circle in the figure, which has at most one incoming CF edge and at most one outgoing CF edge. Conditional vertices come in pairs, represented by diamond and triangle in the figure, which separately denote the *entry* vertex v_k^{en} and *exit* vertex v_k^{ex} of a *conditional block*. We formulate two types of conditional blocks as follows.

- **`If-else` block** B_{if} contains two *branches* B_1 and B_2, which are disjoint with each other. Each branch B_i $(i = 1, 2)$ is also a subgraph of τ_k with a single entry vertex and a single exit vertex. The entry vertex of B_i is pointed by a CF edge from B_{if}'s entry vertex v_k^{en}. The exit vertex of B_i points to B_{if}'s exit vertex v_k^{ex} via a CF edge. Fig. 2 shows an `if-else` block with entry vertex v_1^3, exit vertex v_1^6 and two branches: v_1^4 and v_1^5.
- **`Loop` block** B_{lp} contains a body B, which is a subgraph of τ_k with a single entry vertex and a single exit vertex. The entry vertex v_k^{en} of B_{lp} has two outgoing CF edges: one points to B's entry vertex, and the other one points to B_{lp}'s exit vertex v_k^{ex}. v_k^{en} is pointed by B's exit vertex via a CF edge, which is called as the *back* edge of B_{lp}. Fig. 2 shows a `loop` block with an entry vertex v_1^2, an exit vertex v_1^7 and a body that contains vertices from v_1^3 to v_1^6. A `loop` block B_{lp} with loop bound K can be *unrolled* into a directed acyclic (sub)graph as shown in Fig. 3, where the body B of B_{lp} is duplicated for K times. In practice, we do not unroll any loops in a program. We just use this concept for illustration purposes.

A vertex v_k^i may be contained in several (nested) conditional blocks, we say v_k^i is *closely* contained in the innermost block among all these nested blocks. Fig. 2 shows that the `if-else` block and the `loop` block both contain v_1^4, which is only closely contained in the `if-else` block.

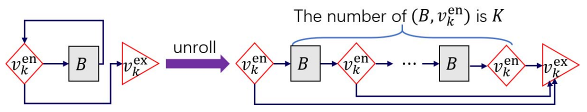


Fig. 3: illustration for unrolling a `loop` block.

Inter-task structure. Recall that the whole OpenMP task system $\mathcal{T}$ can be represented by a digraph, which contains the CFG of each task in $\mathcal{T}$. The inter-task edges of $\mathcal{T}$'s digraph can only connect the task and its child tasks. For the sake of convenience, instead of the whole digraph of $\mathcal{T}$, we separately draw the CFG for each task τ_k of $\mathcal{T}$ (with τ_k's child tasks, which are represented by rectangles in the figure). We formulate two types of inter-task edges as follows.

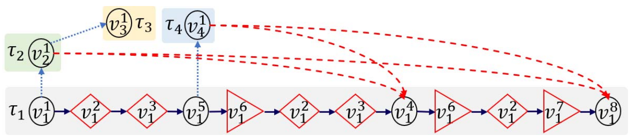
- **Task creation (TC) edge**, denoted by dotted-line arrows in the figure. A vertex v_k^i is called the `creation` vertex if v_k^i represents the `task` directive that creates a child task τ_l

of τ_k. There is a TC edge from the `creation` vertex v_k^i to the source vertex v_l^{src} of τ_l. Fig. 2 shows a `creation` vertex v_1^1, which creates τ_2. Consequently, (v_1^1, v_2^1) is a TC edge.

- **Task wait (TW) edge**, denoted by dashed-line arrows in the figure. A vertex v_k^j is called the `wait` vertex if v_k^j represents the `taskwait` directive in the code region of τ_k. For any task τ_l whose `creation` vertex v_k^i is a predecessor of v_k^j, there is a TW edge from the sink vertex v_l^{snk} of τ_l to the `wait` vertex v_k^j. As shown in Fig. 2, v_1^4 is a wait vertex. There is a TW edge from v_2^1 to v_1^4, in the sense that τ_2's `creation` vertex v_1^1 is the predecessor of v_1^4. Moreover, there is a TW edge from v_4^1 to v_1^4 since v_1^5 is a predecessor of v_1^4 (The reason is that there is a path from v_1^5 to v_1^4 via the back edge (v_1^6, v_1^2) of the `loop` block.)

Execution Model. We note that the digraph of $\mathcal{T}$ may have cycles due to `loop` structures. We unroll all `loop` blocks of $\mathcal{T}$ from the innermost loop to the outermost loop, and obtain the DAG of $\mathcal{T}$, based on which the *execution flow* of $\mathcal{T}$ is defined as follows.

Definition 1 (Execution Flow)**.** *The execution flow (EF) ϵ of $\mathcal{T}$ (written as $\epsilon \vdash \mathcal{T}$) is a subgraph of $\mathcal{T}$'s DAG such that the source vertex v_1^{src} of the main task τ_1 must belong to ϵ, and for any vertex v_k^i that has been involved in ϵ, if v_k^i is an entry vertex of a conditional block, only one of its immediate successors is involved in ϵ. Otherwise, all of its immediate successors (via CF edges and TC edges) are involved in ϵ. An edge (v_k^i, v_l^j) of $\mathcal{T}$'s DAG is involved in ϵ only if its ending points v_k^i and v_l^j are both involved in ϵ.*



τ_2 v_2^1
v_3^1 τ_3
τ_4 v_4^1
τ_1 v_1^1 v_1^2 v_1^3 v_1^5 v_1^6 v_1^2 v_1^3 v_1^4 v_1^6 v_1^2 v_1^7 v_1^8

Fig. 4: The EF ϵ of the OpenMP task system $\mathcal{T}$ in Fig. 2.

Fig. 4 shows an example EF ϵ of $\mathcal{T}$ in Fig. 2, which starts with the source vertex v_1^1 of τ_1. The task τ_2, τ_3 and τ_4 (as well as their corresponding TC edges) are involved in ϵ since their `creation` vertices (e.g, v_1^1, v_2^1 and v_1^5) are involved in ϵ. The body of the `loop` block is executed twice in ϵ, and thus, the vertices in the `loop` block are duplicated. The TW edges between τ_1's child task and the `wait` vertices of τ_1 are also added into ϵ.

The execution of $\mathcal{T}$ is equivalent to travel an EF ϵ of $\mathcal{T}$, which starts with the source vertex v_1^{src}. A vertex becomes *eligible* to be executed only when its predecessors (in ϵ) are all completed. When a vertex is completed, all its successors (in ϵ) should be executed. In Fig. 4, the vertex v_1^4 is eligible if the vertices v_1^3, v_4^1 and v_2^1 are completed. According to Sec. III, we schedule the EF ϵ on m threads under work-conserving algorithms, in the sense that a thread always tries to execute vertices of ϵ, and no threads will be left idle if there are vertices eligible for execution.

V. Response Time Bounds

For any EF ϵ of $\mathcal{T}$, we use $R(\epsilon)$ to denote the worst-case response time (WCRT) of ϵ, which equals the maximum time taken to execute all vertices in ϵ on m threads. The WCRT $R(\mathcal{T})$ of $\mathcal{T}$ is defined as follows.

$$R(\mathcal{T}) = \max_{\epsilon \vdash \mathcal{T}} R(\epsilon) \tag{1}$$

It is difficult to precisely compute the WCRT $R(\mathcal{T})$ due to *timing anomalies* [25], i.e., for any EF ϵ of $\mathcal{T}$, the WCRT $R(\epsilon)$ is not always led by executing each vertex in ϵ with the worst-case execution time. Instead, we aim to compute an upper bound of WCRT $R(\mathcal{T})$ by using two important parameters defined as follows.

Definition 2 (length)**.** *For any EF ϵ of $\mathcal{T}$, we use $len(\epsilon)$ to denote the length of the longest path in ϵ. The length of $\mathcal{T}$ is denoted as $len(\mathcal{T}) = \max_{\epsilon \vdash \mathcal{T}} len(\epsilon)$.*

Definition 3 (volume)**.** *For any EF ϵ of $\mathcal{T}$, we use $vol(\epsilon)$ to denote the total execution time of the vertices in ϵ. The volume of $\mathcal{T}$ is denoted as $vol(\mathcal{T}) = \max_{\epsilon \vdash \mathcal{T}} vol(\epsilon)$.*

For example, by assuming that all conditional vertices in Fig. 4 have zero execution time, and all non-conditional vertices have unit execution time. The length of ϵ is 5 , and the longest path of ϵ is $\pi = (v_1^1, v_1^2, v_1^3, v_1^5, v_4^1, v_1^4, v_1^6, v_1^2, v_1^7, v_1^8)$. The volume of ϵ equals 7. The following theorem gives an upper bound of WCRT $R(\mathcal{T})$.

Theorem 1. *Under work-conserving algorithms, tasks of $\mathcal{T}$ executed on m threads have the WCRT bounded by*

$$R(\mathcal{T}) \leq \frac{m-1}{m} len(\mathcal{T}) + \frac{1}{m} vol(\mathcal{T}) \tag{2}$$

Proof. Since EF ϵ is a DAG, when ϵ is scheduled by work-conserving algorithms on m threads, the WCRT $R(\epsilon)$ can be bounded by the Graham bound [11]:

$$R(\epsilon) \leq \frac{m-1}{m} len(\epsilon) + \frac{vol(\epsilon)}{m} \tag{3}$$

and by (1), and moreover, since $len(\mathcal{T}) \geq len(\epsilon)$ and $vol(\mathcal{T}) \geq vol(\epsilon)$, the lemma is proved. □

From Thm. 1, we can separately compute the volume $vol(\mathcal{T})$ and the length $len(\mathcal{T})$ of $\mathcal{T}$. Then we combine these two parameters together by (2), and eventually obtain the upper bound of WCRT $R(\mathcal{T})$. Computing the parameters for cyclic digraph is challenging especially when loops contain if-else components due to the fact that the maximum value of the parameter is not necessarily led by taking the same branch (of the `if-else` block) in different iterations of the `loop` as illustrated in the following example.

Example 1. *We consider the digraph of $\mathcal{T}$ in Fig. 2, and take the length computation for example. We assume that the execution time of each conditional vertex is zero, and the execution time of each non-conditional vertex is 1. The* `loop` *in Fig. 2 cannot be iterated more than two times. There are two possible cases.*

- ***Case 1***. *The* `if-else` *block nested in the* `loop` *block always takes the same branch in each iteration as shown in Fig. 5. The longest path of* ϵ_1 *is* $\pi_1 = (v_1^1, v_2^1, v_1^4, v_1^6, v_1^2, v_1^3, v_1^4, v_1^6, v_1^2, v_1^7, v_1^8)$ *and the longest path of* ϵ_2 *is* $\pi_2 = (v_1^1, v_1^2, v_1^3, v_1^5, v_1^6, v_1^2, v_1^3, v_1^5, v_4^1, v_1^8)$*, both of which have the same length* 5.

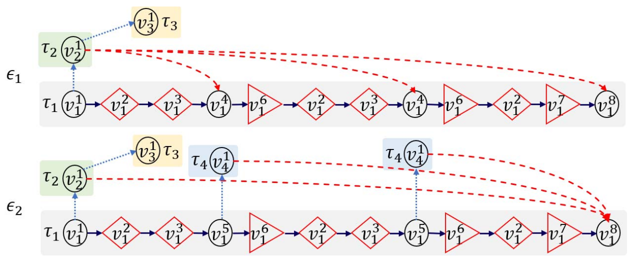


Fig. 5: The EFs ϵ_1 and ϵ_2 with `if-else` block taking the same branch in each iteration of the `loop` block.

- ***Case 2***. *The* `if-else` *block nested in the* `loop` *block takes different branch in each iteration. We obtain the longest path* $\pi^* = (v_1^1, v_2^1, v_1^4, v_1^6, v_1^2, v_1^3, v_1^5, v_4^1, v_1^8)$ *of* $\mathcal{T}$ *with length 6 when the digraph of* $\mathcal{T}$ *is executed as the EF* ϵ_3 *that takes the branch* v_1^4 *in the first iteration, and takes the branch* v_1^5 *in the second iteration as shown in Fig. 6.*

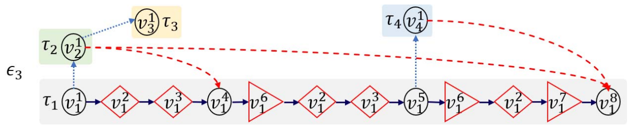


Fig. 6: The EF ϵ_3 with `if-else` block taking different branch in each iteration of the `loop` block.

In sum, we know that the length of $\mathcal{T}$ *in Fig. 2 is not led by taking the same branch in each iteration of the* `loop` *block.*

The above example indicates that we cannot exactly compute the parameters unless the inner-structure of `loop` blocks is carefully investigated. A straightforward way to compute the parameters (e.g., the length and the volume) in (2) is to unroll all `loops` and transform the (cyclic) digraph of $\mathcal{T}$ to an equivalent DAG. Then we enumerate all EFs on $\mathcal{T}$'s DAG, and find the one with the maximum parameter. However, it is impractical since the number of EFs of $\mathcal{T}$ is exponential regarding the number of `loop`/`if-else` blocks. Moreover, simply unrolling `loops` results in a very large DAG model especially in the presence of nested `loops`, which makes the WCRT bound calculation problem computationally intractable.

In the following, we will propose efficient bound computation methods that avoid `loop` unrolling or explicit execution-flow enumeration. More specifically, we first compute the volume $vol(\mathcal{T})$ in Sec. VI, and then compute the length $len(\mathcal{T})$ in Sec. VII. Computation results of these two sections are eventually combined to solve the WCRT bound according to (2). In both of the following two sections, we first propose an approximation method that is based on an intuitive idea and will be used as the baseline algorithm in the evaluation work. Then, by exploring deep insights into $\mathcal{T}$'s hierarchical structure, we propose a more precise method which is a little complicated, but still remains linear-time complexity.

VI. Computing $vol(\mathcal{T})$

A. Approximation Method

A trivial bound for the volume $vol(\mathcal{T})$ is given below.

$$vol(\mathcal{T}) \le \sum_{\tau_k \in \mathcal{T}} tc_{max}(\tau_k) \times \sum_{v_k^i \in \tau_k} tt_{max}(v_k^i) \times c_k^i \tag{4}$$

where $tc_{max}(\tau_k)$ is the maximum number of times that the task τ_k can be created in an EF ϵ of $\mathcal{T}$. $tt_{max}(v_k^i)$ is the maximum number of times that the vertex v_k^i can be traveled in each execution of τ_k. These parameters $tc_{max}(\tau_k)$ and $tt_{max}(v_k^i)$ are calculated by Alg. 1. First, we sort the tasks of $\mathcal{T}$ in the order $\sigma(\mathcal{T})$ such that for any two tasks τ_k and τ_l, τ_l is before τ_k if τ_l is the parent of τ_k. Exploiting the order of $\sigma(\mathcal{T})$, we compute the maximum creation time $tc_{max}(\tau_k)$ for each task $\tau_k \in \mathcal{T}$ (see in Lines 1 to 6). For each task τ_k, we use $\gamma(\tau_k)$ to denote the order in which the vertices of τ_k are sorted topologically (by ignoring all back edges). We compute the maximum traversal times of τ_k's vertices in the order of $\gamma(\tau_k)$ as shown in Lines 7 to 15.

Algorithm 1: Computing parameters in (4).

```
1  for each task $\tau_k$ of $\sigma(\mathcal{T})$ from the first to the last do
2      if $\tau_k$ is the main task of $\mathcal{T}$ then
3          $tc_{max}(\tau_k) := 1$;
4      else
5          //suppose that $\tau_k$ is created by vertex $v_l^j$ of $\tau_l$
6          $tc_{max}(\tau_k) := tt_{max}(v_l^j) \times tc_{max}(\tau_l)$;
7      for each vertex $v_k^i$ of $\gamma(\tau_k)$ from the first to the last do
8          if $v_k^i$ is the entry vertx of a loop block $B_{lp}$ then
9              // denote by $K$ the loop bound of $B_{lp}$;
10             $tt_{max}(v_k^i) := (K + 1)$;
11         else
12             $tt_{max}(v_k^i) := 1$;
13         if $v_k^i$ is contained in some loop blocks then
14             //suppose that $v_k^i$ is closely contained in a loop
                 block $B_{lp}$ with the exit vertex $v_k^{ex}$ and the loop
                 bound $K$
15             $tt_{max}(v_k^i) := tt_{max}(v_k^i) \times tt_{max}(v_k^{ex}) \times K$;
```

Complexity. The parameter computation for each task τ_k costs $O(|V_k|)$, where $|V_k|$ is the number of vertices in τ_k. The complexity of Alg. 1 is $O(N)$, where $N = \sum_{\tau_k \in \mathcal{T}} |V_k|$ is the total number of vertices in all tasks of $\mathcal{T}$.

Pessimism. Formula (4) involves all vertices of $\mathcal{T}$, and requires that each vertex is traveled up to its maximum traversal times. This may not correspond to a feasible EF since the vertices in different branches of the same `if-else` block should be mutually traveled. For example, by assuming that bound of the `loop` block in Fig. 2 is 2, the vertex v_1^5 is traveled twice in an EF ϵ only if the vertex v_1^4 is not traveled in ϵ. By (4), the vertices v_1^5 and v_1^4 are both required to be traveled twice, which clearly violates the loop bound constraint. Formula (4) is pessimistic and cannot compute an exact volume of $\mathcal{T}$.

B. More Exact Method

To compute $vol(\mathcal{T})$ more exactly, we explore deep insights into the hierarchical structure of a task, and propose the definition of syntax tree as follows.

Definition 4 (syntax tree)**.** *For any task τ_k of $\mathcal{T}$, the syntax tree $tr(\tau_k)$ of τ_k is denoted as follows.*

- *The root node of $tr(\tau_k)$ represents τ_k itself.*
- *Each leaf node of $tr(\tau_k)$ corresponds to a vertex of τ_k.*
- *A non-leaf node B of $tr(\tau_k)$ represents a subgraph of τ_k:*
  - *If B represents an `if-else` block B_{if}, it has four child nodes v_k^{en}, v_k^{ex}, B_1 and B_2, which separately represent the entry vertex, the exit vertex and the two branches of B_{if}.*
  - *If B represents a `loop` block B_{lp}, it has three child nodes v_k^{en}, v_k^{ex} and B', which separately represent the entry vertex, the exit vertex and the body of B_{lp}.*
  - *If B represents a sequence of blocks, it has two child nodes B_1 and B_2, which separately represent the first block B_1 of B and the rest blocks of B, i.e., $B_2 = B - B_1$.*

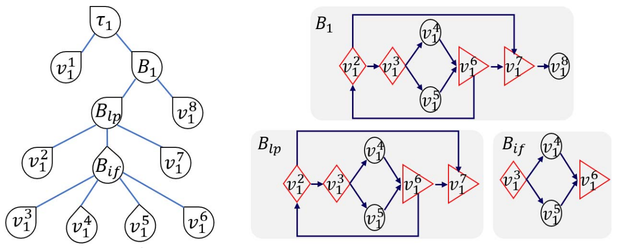


Fig. 7: The syntax tree of τ_1 in Fig. 2.

As shown in Fig. 7, the task τ_1 in Fig. 2 with 8 vertices corresponds to a syntax tree with 12 nodes. For reducing notations, we use the same symbol to denote the tree node and the subgraph represented by the tree node.

Lemma 1. *The number of nodes in $tr(\tau_k)$ is less than $2|V_k|$.*

Proof. As we know that there are at most n non-leaf nodes in a complete binary tree with n leaf nodes. According to Def. 4, a non-leaf node of $tr(\tau_k)$ has at least two child nodes, and thus, the non-leaf nodes of $tr(\tau_k)$ is less than that of the complete binary tree if they have the same number of leaf nodes. Moreover, since each leaf node of $tr(\tau_k)$ corresponds to a vertex of τ_k, there are at most $|V_k|$ non-leaf nodes in $tr(\tau_k)$. This completes the proof. □

Definition 5 (induced digraph)**.** *For any node B of $tr(\tau_k)$ that represents a subgraph B of τ_k, the digraph $G(B)$ induced by B contains not only the vertices in B but also the induced digraphs of the tasks created by the `creation` vertices in B. The volume of the induced digraph $G(B)$ of B is denoted as $vol(B) = \max\{vol(\epsilon_B)|\epsilon_B = G(B) \cap \epsilon$, for any $\epsilon \vdash \mathcal{T}\}$.*

For example, by considering the subgraph B_1 in Fig. 7, its induced digraph $G(B_1)$ contains B_1 itself, the CFG of τ_4 that is created by the vertex v_1^5 in B_1, and the TC and TW edges between the vertices of B_1 and the vertices in τ_4. The volume $vol(B_1)$ of the induced digraph $G(B_1)$ equals 4 by assuming that each non-conditional vertex has an unit execution time and each conditional vertex has zero execution time. The induced graph of the main task τ_1 is the whole digraph of $\mathcal{T}$, and thus, the volume of $\mathcal{T}$ equals that of τ_1's induced digraph, i.e., $vol(\mathcal{T}) = vol(\tau_1)$. Based on this observation, we can compute the volume $vol(\mathcal{T})$ by solving $vol(\tau_1)$. We propose a dynamic programming algorithm to compute $vol(\tau_1)$ as shown in Alg. 2.

Algorithm 2: Computing $vol(\mathcal{T})$.

```
1  for each task $\tau_k$ in $\sigma(\mathcal{T})$ from the last to the first do
2      for each node $B$ in $tr(\tau_k)$ from leaves to the root do
3          if $B$ is a leaf node representing vertex $v_k^i$ then
4              $vol(B) := c_k^i$;
5              if $v_k^i$ is a creation vertex then
6                  $vol(B) := vol(B) + vol(\tau_l)$; // $v_k^i$ creates $\tau_l$;
7          else if $B$ is the sequence of blocks $(B_1, B_2)$ then
8              $vol(B) := vol(B_1) + vol(B_2)$;
9          else if $B$ is an if-else block then
10             $vol(B) := c_k^{en} + c_k^{ex} + \max\{vol(B_1), vol(B_2)\}$;
11         else if $B$ is a loop block with body $B'$ then
12             $vol(B) := (K+1) \times c_k^{en} + c_k^{ex} + K \times vol(B')$;
13 return $vol(\mathcal{T}) := vol(\tau_1)$;
```

As shown in Line 1 of Alg. 2, by exploiting the reversed order of $\sigma(\mathcal{T})$, we compute the volume $vol(\tau_k)$ of the digraph induced from each task $\tau_k \in \mathcal{T}$ in an iterative way, i.e., for any task τ_k and its child task τ_l, the volume $vol(\tau_l)$ computed for τ_l can be used for the volume computation of (the induced digraph of) τ_k. For each task τ_k, we travel its syntax tree $tr(\tau_k)$ from leaves to root (Line 2), and compute the volume $vol(B)$ for each node B in $tr(\tau_k)$. If B is a leaf of $tr(\tau_k)$ that represents a vertex v_k^i of τ_k, we compute $vol(B)$ by distinguishing whether v_k^i is a creation vertex or not (see in Lines 3 to 6,). Otherwise, B represents a non-leaf node of $tr(\tau_k)$. There are three possibilities. As shown in Lines 7 to 8, if B corresponds to a sequence of blocks (B_1, B_2), the volume $vol(B)$ is the summation of $vol(B_1)$ and $vol(B_2)$. If B corresponds to an `if-else` block with branches B_1 and B_2, the branch of B with the larger volume contributes to $vol(B)$ as shown in Line 10. If B corresponds to a `loop` block B_{lp} with the body B' and loop bound K, we know that the entry vertex v_k^{en} of B_{lp} can be executed at most $K+1$ times and the body B' of B_{lp} can be executed at most K times before jumping out of the loop. It indicates that $vol(B)$ is calculated by the equation in Line 12. Alg. 2 returns the volume $vol(\tau_1)$ of the main task τ_1's induced digraph as the volume of $\mathcal{T}$ as shown in Line 13.

Complexity. The volume computation for each task τ_k costs $O(2|V_k|)$ according to Lem. 1. The complexity of Alg. 2 is $O(2N)$, recalling that N denotes the total number of vertices in all tasks of $\mathcal{T}$.

VII. Computing $len(\mathcal{T})$

A. Approximation Method

We solve an upper bound of $len(\mathcal{T})$ by Alg. 3. For each task τ_k and each node B of τ_k's syntax tree $tr(\tau_k)$, we use $len(B)$ to denote the length of the longest path in the induced digraph $G(B)$ of the block B, and we denote by $len^u(B)$ the upper bound of $len(B)$. For each task τ_k in $\mathcal{T}$, we compute $len^u(\tau_k)$ of the induced digraph $G(\tau_k)$, and the length $len(\mathcal{T})$ is bounded by $len^u(\tau_1)$ as shown in Line 13. The computation of $len^u(\tau_k)$ relies on the computation results of τ_k's child tasks. Therefore, we exploit the reverse order of $\sigma(\mathcal{T})$ (Line 1), and ensure that before computing $len^u(\tau_k)$ of the task τ_k, the parameters $len^u(\tau_l)$ of τ_k's child tasks τ_l have been computed. For each task τ_k, we explore the nodes of τ_k's syntax tree $tr(\tau_k)$ from leaf nodes to the root node. For each node B of $tr(\tau_k)$, we bound the length of B's induced digraph $G(B)$ by considering the following two cases.

Algorithm 3: Computing the upper bound of $len(\mathcal{T})$.

```
1  for each task $\tau_k$ in $\sigma(\mathcal{T})$ from the last to the first do
2      for each node $B$ in $tr(\tau_k)$ from leaves to the root do
3          if $B$ is a leaf node of $tr(\tau_k)$ representing vertex $v_k^i$ then
4              $len^u(B) := c_k^i$;
5              if $v_k^i$ is creation vertex that creates $\tau_l$ then
6                  $len^u(B) := len^u(B) + len^u(\tau_l)$;
7          else if $B$ is the sequence of blocks $(B_1, B_2)$ then
8              $len^u(B) := len^u(B_1) + len^u(B_2)$;
9          else if $B$ is an if-else block then
10             $len^u(B) := c_k^{en} + c_k^{ex} + \max\{len^u(B_1), len^u(B_2)\}$;
11         else if $B$ is a loop block with body $B'$ then
12             $len^u(B) := (K+1) \times c_k^{en} + c_k^{ex} + K \times len^u(B')$;
13 return $len^u(\mathcal{T}) := len^u(\tau_1)$;
```

- If B is a leaf node of $tr(\tau_k)$ (see in Line 3), we assume that B represents a vertex v_k^i of τ_k, and there are two possibilities. 1) v_k^i is a creation vertex that creates the child task τ_l of τ_k. The length of B is bounded by the summation of c_k^i and the upper bound $len^u(\tau_l)$ of the length related to τ_l as shown in Line 6. 2) v_k^i is not a creation vertex, and in this case, the length of B is simply bounded by c_k^i as shown in Line 4.
- Otherwise, a non-leaf node B has three possibilities. 1) B represents a sequence of blocks. The first block of B is denoted as B_1, and we let $B_2 = B - B_1$. The length of B is bounded by the summation of $len^u(B_1)$ and $len^u(B_2)$ as shown in Line 8. 2) B represents an `if-else` block B_{if}. The branch of B with larger length bound is used to derive the upper bound of $len(B)$ as shown in Line 10. 3) B represents a `loop` block B_{lp}. The upper bound $len^u(B)$ is calculated in Line 12 as we know that the entry vertex v_k^{en} of B_{lp} is traveled at most $K+1$ times, and the path of B_{lp}'s body B' is traveled at most K times.

Complexity. We observe that Alg. 3 is obtained by slightly modifying Alg. 2, and these two algorithms have the same computation complexity. According to Lem. 1, Alg. 3 computes the upper bound of $len(\mathcal{T})$ within $O(2N)$, where N is the total number of vertices in all tasks of $\mathcal{T}$.

Pessimism. Alg. 3 cannot solve the length $len(\mathcal{T})$ exactly. We take the computation of $len^u(B_{lp})$ for the `loop` block B_{lp} in Fig. 2 as an example. We assume that all conditional vertices have zero execution time, and all non-conditional vertices have unit execution time. The loop bound of B_{lp} in Fig. 2 is 2. According to Lines 11 and 12, the length bound $len^u(B_{lp})$ equals $2 \times len^u(B_{if})$, where B_{if} is the `if-else` block inside B_{lp}. The longest path of B_{if} is (v_1^3, v_1^5, v_4^1) with the length 2. Therefore, $len^u(B_{lp}) = 4$. Actually, the longest path of B_{lp} is $(v_1^2, v_1^3, v_1^5, v_4^1, v_1^4, v_1^6, v_1^2, v_1^7)$ with length 3, which is less than the length bound $len^u(B_{lp})$ computed by Alg. 3. By using the computation result $len^u(B_{lp})$, we further derive the length bound $len^u(B_1) = 5$ for the block B_1 (as shown in Fig. 7) according to Lines 7 and 8. Moreover, the length bound $len^u(v_1^1)$ of the digraph $G(v_1^1)$ induced by v_1^1 equals 3 according to Lines 5 and 6. The length bound $len^u(\tau_1)$ for the main task τ_1 equals $len^u(v_1^1) + len^u(B_1) = 8$, according to Lines 7 and 8. According to Line 13, the length bound of $\mathcal{T}$ returned by Alg. 3 equals 8. As shown in Case 2 of Example 1, the longest path of $\mathcal{T}$ has the length 6, which is less than the one calculated by Alg. 3.

B. More Exact Method

From above section, we know that for any block B of τ_k, it is not sufficient to exactly derive the length of $\mathcal{T}$ if we only know the length of the longest path of B's induced digraph $G(B)$. In this sub-section, we propose a dynamic programming algorithm to compute the length $len(\mathcal{T})$ more exactly. To this end, we explore deep insights about the inner structure of B's induced digraph $G(B)$, and introduce six types of paths that travel the vertices of $G(B)$ with an entry vertex v_k^{en} and an exit vertex v_k^{ex} as follows.

- $\Pi_{en}^*(B)$: the set of paths that start with the entry vertex v_k^{en} of B. For example, the path (v_1^3, v_1^4) is a path in $\Pi_{en}^*(B_{if})$ for the `if-else` block B_{if} in Fig. 2.
- $\Pi_{wt}^*(B)$: the set of paths that start with a wait vertex of B. For example, the path $(v_1^4, v_1^6, v_1^2, v_1^3, v_1^5)$ is a path in $\Pi_{wt}^*(B_{lp})$ for the `loop` block B_{lp} in Fig. 2.
- $\Pi_{en}^{ex}(B)$: the set of paths that start with the entry vertex v_k^{en} of B and end at the exit vertex v_k^{ex} of B. For example, the path (v_1^3, v_1^5, v_1^6) is a path in $\Pi_{en}^{ex}(B_{if})$ for the `if-else` block B_{if} in Fig. 2.
- $\Pi_{en}^{ct}(B)$: the set of paths that start with the entry vertex v_k^{en} of B and end at the sink vertex of a child task of τ_k. For example, the path (v_1^3, v_1^5, v_4^1) is a path in $\Pi_{en}^{ct}(B_{if})$ for the `if-else` block B_{if} in Fig. 2.
- $\Pi_{wt}^{ex}(B)$: the set of paths that start with a wait vertex of B and end at the exit vertex v_k^{ex} of B. For example, the path (v_1^4, v_1^6) is a path in $\Pi_{wt}^{ex}(B_{if})$ for the `if-else` block B_{if} in Fig. 2.
- $\Pi_{wt}^{ct}(B)$: the set of paths that start with a wait vertex of B and end at the sink vertex of a child task of τ_k. For example,

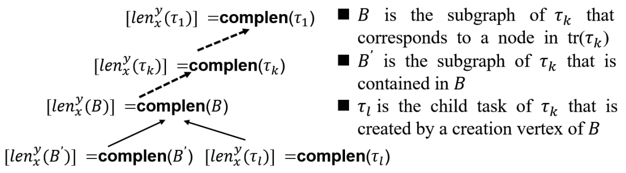
the path $(v_1^4, v_1^6, v_1^2, v_1^3, v_1^5, v_4^1)$ is a path in $\Pi_{wt}^{ct}(B_{lp})$ for the `loop` block B_{lp} in Fig. 2.

The longest path π of $\mathcal{T}$ must travel the paths of the above path sets if π travels B. We illustrate this by considering two possible cases. (1) If π enters into block B from the entry of B, π contains the sub-path belonging to one of three path sets Π_{en}^y, where $y = \{*, ex, ct\}$. (2) Otherwise, π must enter into B from a `wait` vertex (via `wait` edges), and in this case, π contains the sub-path belonging to one of three path sets Π_{wt}^y, where $y = \{*, ex, ct\}$. For any path set $\Pi_x^y(B)$ (where $x \in \{en, wt\}$ and $y = \{*, ex, ct\}$), we use $len_x^y(B)$ to denote the length of the longest path in $\Pi_x^y(B)$, i.e., $len_x^y(B) = \max_{\pi' \in \Pi_x^y(B)} len(\pi')$. For any task $\tau_k \in \mathcal{T}$ and for any node B of $tr(\tau_k)$, we use the procedure **complen(B)** to compute all lengths $len_x^y(B)$ of $G(B)$ where $x = \{en, wt\}$ and $y = \{ex, ct, *\}$.

Algorithm 4: Framework for computing $len(\mathcal{T})$.

```
1 for each task $\tau_k$ in $\sigma(\mathcal{T})$ from the last to the first do
2     for each node $B$ in $tr(\tau_k)$ from leaves to the root do
3         complen($B$);
4 return $len(\mathcal{T}) := len(\tau_1)$;
```

Alg. 4 gives a framework to solve the length $len(\mathcal{T})$ of $\mathcal{T}$. For each task τ_k and for each node B of $tr(\tau_k)$, the length computation procedure $complen(B)$ uses the length computation results of B's child nodes and τ_k's child tasks (Lines 1 to 3). More precisely, the precedence order between the computation procedures is described in Fig. 8.



$[len_x^y(\tau_1)]$ =**complen**(τ_1)
$[len_x^y(\tau_k)]$ =**complen**(τ_k)
$[len_x^y(B)]$ =**complen**(B)
$[len_x^y(B')]$ =**complen**(B')
$[len_x^y(\tau_l)]$ =**complen**(τ_l)
■ B is the subgraph of τ_k that corresponds to a node in tr(τ_k)
■ B' is the subgraph of τ_k that is contained in B
■ τ_l is the child task of τ_k that is created by a creation vertex of B

Fig. 8: The precedence order of computation functions.

By letting B' be the child node of B, the computation of **complen**(B) relies on the computation results of **complen**(B'). Moreover, by using the lengths computed by **complen**(B), we can eventually compute the lengths $len_x^y(\tau_k)$ for the task τ_k whose syntax tree contains node B. Besides the lengths computed by **complen**(B), the computation of **complen**(τ_k) also needs the computation results of **complen**(τ_l) where τ_l is the child task of τ_k. When every task completes its length computation, we can derive the length of $\mathcal{T}$ as $len(\mathcal{T}) = len_{en}^*(\tau_1)$ for τ_1 is the main task of $\mathcal{T}$ (Line 4). The key point of Alg. 4 is how to implement $complen(B)$ for a given node B of $tr(\tau_k)$, which is described below in more details.

Procedure of complen(B)

In the following, we propose a dynamic programming algorithm to compute these longest path lengths $len_x^y(B)$ for $x \in \{en, wt\}$ and $y = \{*, ex, ct\}$. There are four possible cases below.

Case 1. B is a vertex v_k^i of τ_k, i.e., $B = v_k^i$. The lengths $len_{en}^y(B)$ (for $y = \{*, ex, ct\}$) are calculated as follows.

$$len_{en}^{ex}(v_k^i) = c_k^i \tag{5}$$

$$len_{en}^{ct}(v_k^i) = \begin{cases} c_k^i + len_{en}^{ex}(\tau_l) & v_k^i \text{ creates } \tau_l, \\ -\infty & \text{else.} \end{cases} \tag{6}$$

$$len_{en}^{*}(v_k^i) = \begin{cases} c_k^i + len_{en}^{*}(\tau_l) & v_k^i \text{ creates } \tau_l, \\ c_k^i & \text{else.} \end{cases} \tag{7}$$

and the lengths $len_{wt}^y(v_k^i)$ (for $y = \{*, ex, ct\}$) are calculated as follows.

$$len_{wt}^{y}(v_k^i) = \begin{cases} len_{en}^{y}(v_k^i) & v_k^i \text{ is a wait vertex,} \\ -\infty & \text{else.} \end{cases} \tag{8}$$

Lemma 2. *For any vertex v_k^i of τ_k, the lengths $len_x^y(v_k^i)$ is correctly computed by (5) to (8).*

Proof. Since there is a single path in $\Pi_{en}^{ex}(v_k^i)$ which only contains vertex v_k^i, the length $len_{en}^{ex}(v_k^i)$ equals c_k^i, and we derive Formula (5). Formula (6) and (7) separately compute $len_{en}^{ct}(v_k^i)$ and $len_{en}^{*}(v_k^i)$ by distinguishing whether v_k^i is a creation vertex or not. If this is the case, we assume that τ_l is the child task of τ_k which is created by v_k^i. The path of $\Pi_{en}^{ct}(v_k^i)$ begins with v_k^i and travels a path in $\Pi_{en}^{ex}(\tau_l)$ related to τ_l. It indicates $len_{en}^{ct}(v_k^i) = c_k^i + len_{en}^{ex}(\tau_l)$. Similarly, the path of $\Pi_{en}^{*}(v_k^i)$ begins with v_k^i and travels a path in $\Pi_{en}^{*}(\tau_l)$, which indicates $len_{en}^{*}(v_k^i) = c_k^i + len_{en}^{*}(\tau_l)$. Otherwise, v_k^i does not create task. In this case, there is no path belonging to $\Pi_{en}^{ct}(v_k^i)$, and thus, we define the length $len_{en}^{ct}(v_k^i)$ as $-\infty$. Moreover, the path in $\Pi_{en}^{*}(v_k^i)$ only travels v_k^i, and thus, $len_{en}^{*}(v_k^i) = c_k^i$. Formula (8) computes $len_{wt}^y(v_k^i)$ by distinguishing whether v_k^i is a wait vertex or not. If this is the case, the path in $\Pi_{wt}^y(v_k^i)$ begins with v_k^i, and thus, the length $len_{wt}^y(v_k^i)$ equals $len_{en}^y(v_k^i)$. Otherwise, there is no path in $\Pi_{wt}^y(v_k^i)$, and thus, we define $len_{wt}^y(v_k^i)$ as $-\infty$. ☐

Case 2. B represents a sequence of blocks, and without loss of generality, we use B_1 to denote the first block of B, and let $B_2 = B - B_1$ represent the rest blocks of B. We calculate the lengths $len_x^y(B)$ (for $x = \{en, wt\}$ and $y = \{*, ex, ct\}$) as follows.

$$\begin{aligned} len_x^y(B) = \max\{&len_x^{ex}(B_1) + len_{en}^y(B_2), len_x^y(B_1), \\ &len_x^{ct}(B_1) + len_{wt}^y(B_2), len_x^y(B_2)\} \end{aligned} \tag{9}$$

Lemma 3. *For any block $B = (B_1, B_2)$, the lengths $len_x^y(B)$ is correctly computed by (9).*

Proof. The path π in $\Pi_x^y(B)$ has two possibilities. 1) π only passes one of $G(B_1)$ and $G(B_2)$, i.e., $\pi \in \Pi_x^y(B_1) \cup \Pi_x^y(B_2)$. It indicates that $len_x^y(B) \geq \max\{len_x^y(B_1), len_x^y(B_2)\}$. 2) π passes both $G(B_1)$ and $G(B_2)$, and can be divided into two parts: $\pi = (\pi_1, \pi_2)$, where π_1 is the path which only travels $G(B_1)$, and π_2 is the path which only travels $G(B_2)$. There are two sub-cases: a) π_1 ends at the exit vertex of B_1, i.e., $\pi_1 \in \Pi_x^{ex}(B_1)$. since there is a control flow edge from the exit vertex of B_1 to the entry vertex of B_2, and in order to guarantee the connectivity between π_1 and π_2, π_2 must begin with the entry vertex of B_2, i.e., $\pi_2 \in \Pi^y_{en}(B_2)$. It indicates that $len^y_x(B) \geq len^{ex}_x(B_1) + len^y_{en}(B_2)$. b) π_1 ends at the sink vertex of a child task τ_l that is created by a vertex in B_1, i.e., $\pi_1 \in \Pi^{ct}_x(B_1)$. Since only wait vertices of B_2 can be connected to the sink vertex of τ_l (via TW edges), to guarantee the connectivity between π_1 and π_2, π_2 must begin with a wait vertex in B_2, i.e., $\pi_2 \in \Pi^y_{wt}(B_2)$ It indicates that $len^y_x(B) \geq len^{ct}_x(B_1) + len^y_{wt}(B_2)$. In sum, we can derive (9). □

Case 3. B represents an `if-else` block which has an entry vertex v^{en}_k, an exit vertex v^{ex}_k and two branches B_1 and B_2. We calculate the lengths $len^y_x(B)$ (for $x = \{en, wt\}$ and $y = \{*, ex, ct\}$) as follows.

$$\begin{aligned}
len^{ex}_{en}(B) &= c^{en}_k + c^{ex}_k + L^{ex}_{en} && (10)\\
len^{ct}_{en}(B) &= c^{en}_k + L^{ct}_{en} && (11)\\
len^{*}_{en}(B) &= \max\{len^{ex}_{en}(B), c^{en}_k + L^{*}_{en}\} && (12)\\
len^{ex}_{wt}(B) &= L^{ex}_{wt} + c^{ex}_k && (13)\\
len^{ct}_{wt}(B) &= L^{ct}_{wt} && (14)\\
len^{*}_{wt}(B) &= \max\{L^{*}_{wt}, len^{ex}_{wt}(B)\} && (15)
\end{aligned}$$

where $L^y_x = \max\{len^y_x(B_1), len^y_x(B_2)\}$.

Lemma 4. *For any `if-else` block B, the lengths $len^y_x(B)$ is correctly computed by (10) to (15).*

Proof. Without loss of generality, we assume that branch B_1 has the larger length $len^y_x(B_1)$ than branch B_2, i.e., $len^y_x(B_1) \geq len^y_x(B_2)$ for each $x = \{en, wt\}$ and $y = \{ex, ct, *\}$. The longest path π in $\Pi^{ex}_{en}(B)$ begins with the entry vertex v^{en}_k of B and ends at the exit vertex v^{ex}_k of B. According to the semantics of an `if-else` block, π can only travel one of the two branches of B. As π is the longest path in $\Pi^{ex}_{en}(B)$, it must travel the longest path in $\Pi^{ex}_{en}(B_1)$, recalling that we have assumed that $len^{ex}_{en}(B_1) \geq len^{ex}_{en}(B_2)$. Thus, the length of π equals $c^{en}_k + len^{ex}_{en}(B_1) + c^{ex}_k$, which indicates (10).

The path in $\Pi^{ct}_{en}(B)$ begins with v^{en}_k and travels the longest path in $\Pi^{ct}_{en}(B_1)$, which indicates (11).

The path π in $\Pi^{*}_{en}(B)$ may have two possibilities. 1) π begins with v^{en}_k and ends at v^{ex}_k, i.e., $\pi \in \Pi^{ex}_{en}(B)$, which indicates $len^{*}_{en}(B) \geq len^{ex}_{en}(B)$. 2) π begins with v^{en}_k and only travels the induced digraph $G(B_1)$ of B_1, and does not travel v^{ex}_k. In this case, π can be represented as $\pi = (v^{en}_k, \pi')$, where π' is the longest path of $G(B_1)$, i.e., $\pi' \in \Pi^{*}_{en}(B_1)$. It indicates that $len^{*}_{en}(B) \geq c^{en}_k + len^{*}_{en}(B_1)$. In sum, we can derive (12).

As the conditional vertex v^{en}_k cannot be a wait vertex, the path in $\Pi^{ex}_{wt}(B)$ should begin with a wait vertex of a branch of B, and ends at v^{ex}_k. It indicates (13). The path in $\Pi^{ct}_{wt}(B)$ only travels the induced digraph of the branch of B, which indicates (14). The path π in $\Pi^{*}_{wt}(B)$ may have two possible cases. 1) π does not end at v^{ex}_k, and in this case, $\pi \in \Pi^{*}_{wt}(B_1)$. 2) π ends at v^{ex}_k, and in this case, $\pi \in \Pi^{ex}_{wt}(B)$. In sum, we can derive (15). □

Case 4. B represents a `loop` block which has an entry vertex v^{en}_k, an exit vertex v^{ex}_k and a body B'. The loop bound of B is K. We calculate lengths $len^y_x(B)$ of B (for $x = \{en, wt\}$ and $y = \{*, ex, ct\}$) as follows. According to Line 2 of Alg. 4, lengths $len^y_x(B')$ of B' must be calculated before the computation of lengths $len^y_x(B)$.

$$\begin{aligned}
len^{ex}_{en}(B) &= c^{en}_k + c^{ex}_k + \max\{\Gamma_1, \Gamma_2\} && (16)\\
len^{ct}_{en}(B) &= c^{en}_k + len^{ct}_{en}(B') + \max\{\Gamma_3, \Gamma_4\} && (17)\\
len^{*}_{en}(B) &= \max\{len^{ex}_{en}(B), len^{ct}_{en}, A_1, A_2\} && (18)\\
len^{ex}_{wt}(B) &= len^{ex}_{wt}(B') + \max\{\Gamma_3, \Gamma_4\} && (19)\\
len^{ct}_{wt}(B) &= len^{ct}_{wt}(B') + \max\{\Gamma_3, \Gamma_4\} && (20)\\
len^{*}_{wt}(B) &= \max\{len^{ex}_{wt}(B), len^{ct}_{wt}(B), A_3\} && (21)
\end{aligned}$$

where

$$\begin{aligned}
\alpha &= c^{en}_k + len^{ex}_{en}(B')\\
\beta &= len^{ct}_{wt}(B')\\
\gamma &= c^{en}_k + len^{ct}_{en}(B') + len^{ex}_{wt}(B')\\
A_1 &= c^{en}_k + len^{*}_{en}(B') + \max\{\Gamma_3, \Gamma_4\}\\
A_2 &= c^{en}_k + len^{ct}_{en}(B') + len^{*}_{wt}(B') + \max\{\Gamma_5, \Gamma_6\}\\
A_3 &= len^{*}_{wt}(B') + \max\{\Gamma_3, \Gamma_4\}\\
\Gamma_1 &= \alpha K\\
\Gamma_2 &= \gamma\lfloor\frac{K}{2}\rfloor + \{\frac{K}{2}\}\max\{\alpha, \beta\}\\
\Gamma_3 &= \alpha(k-1)\\
\Gamma_4 &= \gamma\lfloor\frac{K-1}{2}\rfloor + \{\frac{K-1}{2}\}\max\{\alpha, \beta\}\\
\Gamma_5 &= \alpha(k-2)\\
\Gamma_6 &= \gamma\lfloor\frac{K-2}{2}\rfloor + \{\frac{K-2}{2}\}\max\{\alpha, \beta\}
\end{aligned}$$

Here we use $\{\cdot\}$ to represent the remainder. In the following, we only prove the correctness of (16), and discuss the correctness of other formulas in Appendix A.

Lemma 5. *For any `loop` block B, the length $len^{ex}_{en}(B)$ is correctly computed by (16).*

Proof. We prove the correctness of (16) as follows. We use an automaton $\mathcal{A}^{ex}_{en}$ to represent the paths of $\Pi^{ex}_{en}(B)$ as shown in Fig. 9.

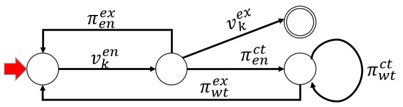


Fig. 9: The automaton $\mathcal{A}^{ex}_{en}$ representing the paths in $\Pi^{ex}_{en}(B)$.

Each edge of $\mathcal{A}^{ex}_{en}$ is associated with a label such as v^{en}_k, v^{ex}_k and π^y_x which denotes the longest path in the path set $\Pi^y_x(B')$ for $x = \{en, wt\}$ and $y = \{*, ex, ct\}$. There is an initial state (pointed by the red arrow) and a final state (marked as a two-layer cycle) in the automaton $\mathcal{A}^{ex}_{en}$. Each path in $\Pi^{ex}_{en}(B)$ can be represented as a path $\mathcal{A}^{ex}_{en}$ from the initial state to the final state. The regular formulation of the paths in $\Pi^{ex}_{en}(B)$ is $v^{en}_k v^{ex}_k (v^{en}_k \pi^{ex}_{en})^{n_1} (\pi^{ct}_{wt})^{n_2} (\pi^{ct}_{en} \pi^{ex}_{wt} v^{en}_k)^{n_3}$, which means that the vertices v_k^{en} and v_k^{ex} are separately traveled once; and the segments $\pi_1 = (v_k^{en}, \pi_{en}^{ex})$, $\pi_2 = \pi_{wt}^{ct}$ and $\pi_3 = (\pi_{en}^{ct}, \pi_{wt}^{ex}, v_k^{en})$ are separately traveled for n_1, n_2 and n_3 times, where n_k (for $k = 1, 2, 3$) is allowed to be 0. Based on these notations, the length $len_{en}^{ex}(B)$ of the longest path in $\Pi_{en}^{ex}(B)$ can be calculated as $len_{en}^{ex}(B) = c_k^{en} + c_k^{ex} + \Gamma$, where Γ is the solution of the following optimization problem.

$$\Gamma = \max\{\alpha n_1 + \beta n_2 + \gamma n_3\} \quad (22)$$

$$s.t. \quad n_1 + n_2 + 2n_3 \leq K \quad (23)$$

where the objective function (22) solves the maximum workload of traveling of the segments π_1, π_2 and π_3. The constraint of (23) ensures the loop bound K. Since $len_{en}^{ct}(B') \geq len_{wt}^{ct}(B')$, we know that $\gamma > \beta$, and there are two cases.

- If $\alpha > \gamma$, the objective function (22) achieves its supremum if $n_1 = K$ and in this case, $n_2 = n_3 = 0$. This indicates that $\Gamma = \Gamma_1$.
- If $\gamma \geq \alpha$, we consider two sub-cases. (1) K is even, the objective function (22) achieves its supremum if $n_3 = \frac{K}{2}$, i.e., $\Gamma = \gamma \frac{K}{2}$. (2) K is odd, the objective function (22) achieves its supremum if $n_3 = \lfloor \frac{K}{2} \rfloor$, and $n_1 = 1$ and $n_2 = 0$ if $\alpha > \beta$ or otherwise, $n_2 = 1$ and $n_1 = 0$. We have $\Gamma = \gamma \lfloor \frac{K}{2} \rfloor + \{\frac{K}{2}\} \max\{\alpha, \beta\}$, and thus, $\Gamma = \Gamma_2$.

In sum, we derive (16). □

Complexity. The length computation procedure **complen**(B) implemented in this sub-section should compute 6 parameters $len_x^y(B)$ for $x = \{en, wt\}$ and $y = \{ex, ct, *\}$, and computation of each parameter costs $O(1)$ time. Moreover, the total number of nodes in a syntax tree $tr(\tau_k)$ is $2|V_k|$ according to Lem. 1. Therefore, the length computation for all tasks in $\mathcal{T}$ costs $O(12N)$, where N is the total number of vertices in $\mathcal{T}$.

VIII. Evaluation

We develop both randomly generated task sets and realistic OpenMP programs to evaluate our WCRT computation methods. We implement our methods in C++, and run the code on a PC with an Intel i5-7500 CPU at 3.4GHZ. For each task set, we use R_1 to denote the WCRT bound with the volume and the length that are separately computed by approximation methods in Sec. VI-A and Sec. VII-A. Moreover, we denote by R_2 the WCRT bound with the parameters computed by the methods in Sec. VI-B and Sec. VII-B. In order to show how many times our exact methods can improve the WCRT bound, we evaluate the bound ratio of R_1 to R_2, i.e., $r = \frac{R_1}{R_2}$. We use t_1 to denote the computation time of the approximation method for computing the WCRT bound R_1, and use t_2 to denote the computation time of the method for computing the WCRT bound R_2. The computation time is measured in milliseconds (ms).

A. Randomly Generated Tasks

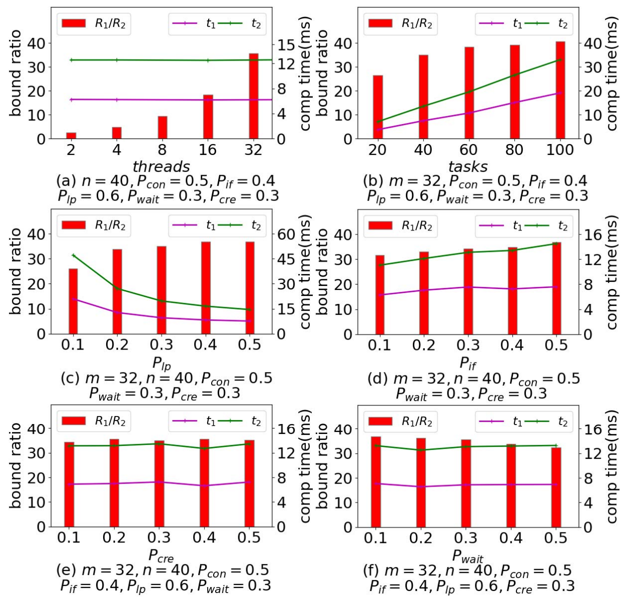
We generate a task system $\mathcal{T}$ that contains n tasks, and use a n-node tree to define the relationship between the tasks of $\mathcal{T}$. Each node of the tree represents a task in $\mathcal{T}$. More specifically, the root of the tree represents the main task τ_1. The leaf node of the tree represents a task that does not have child task. The non-leaf node of the tree represents a task that has child tasks. The edge of the tree indicates the parent-child relation between the corresponding tasks. Here the n-node tree is randomly generated by the Prüfer method [30]: We let T_n denote the set of all possible free trees with n tree nodes. The number of trees in T_n is given by Cayley's celebrated formula $|T_n| = 2^{n-2}$. The Prüfer method first randomly generates a Prüfer string with n elements and then encodes the Prüfer string into a n-node tree, which can generate trees of T_n with equal possibility.

For any task τ_k of $\mathcal{T}$, we assume that τ_k has n_k child tasks, and we randomly generate a CFG of τ_k with about $K = \frac{n_k}{p_{cre}}$ non-conditional vertices, where p_{cre} is the possibility that a non-conditional vertex of τ_k is a `creation` vertex. At the very beginning, we generate a single vertex, and add it into the vertex set V_{new}. In each round, for each newly generated vertex v_k^i in V_{new}, we generate the successor vertex of v_k^i with the possibility $p_{suc} = \max\{0, 1 - \frac{n_{non}}{K}\}$, where n_{con} is the number of non-conditional vertices in the current CFG of τ_k. Moreover, we replace the vertex v_k^i by a conditional block with the possibility p_{cnd}. Furthermore, we let $p_{cnd} = p_{if} + p_{lp}$, where p_{if} is the possibility that a vertex is replaced by an `if-else` block, and p_{lp} is the possibility that a vertex is replaced by a `loop` block. The `if-else` block that is used to replace v_k^i has an entry vertex, an exit vertex and two branches. Each branch contains a single vertex, which is added into the vertex set V_{new}. The `loop` block that is used to replace v_k^i has an entry vertex, an exit vertex and a body, which contains a single vertex v_k^j. We add the newly generated vertex v_k^j into the vertex set V_{new}. The loop bound of a `loop` block is randomly picked in the range $[5, 10]$. Each vertex has the worst-case execution time randomly picked in the range $[1, 10]$. This procedure is repeated until there are (more than) K non-conditional vertices in τ_k. We randomly select n_k non-conditional vertices to be the creation vertices, and each points to a child task of τ_k. Since the total number of non-conditional vertices is about $\frac{n_k}{p_{cre}}$, we know that each conditional vertex is a creation vertex with the possibility p_{cre}. For any conditional vertex v_k^i, if a predecessor of v_k^i creates the child task of τ_k, we let v_k^i be a wait vertex with the possibility p_{wait}.

We conduct experiments with different combinations of parameters in Fig. 10. The values of the configurations are written in the figure caption. For each data point, 1000 random experiments have been run. We observe that our method can significantly improve the WCRT bound, i.e., R_2 is 31.35 times smaller than R_1 on average. Moreover, our methods are very fast, i.e., R_1 can be solved within 8.41 ms and R_2 can be solved within 16.13 ms on average.

As shown in Fig. 10(a), the ratio r of R_1 to R_2 increases exponentially with the increase of the thread number m. According to (2), the WCRT bounds R_1 and R_2 both decrease when m increases. The trend of the ratio r in Fig. 10(a) indicates R_2 decreases much faster than R_1. The computation time t_1 and t_2 is unchanged with the increase of m. This is because that the thread number m does not affect the complexity of our methods.

As shown in Fig. 10(b), the ratio r of R_1 to R_2 increases when the task number n increases. We can conclude that R_2 can tolerant the increase of tasks much better than R_1. The computation time t_1 and t_2 both increase linearly with the increase of n.



(a) $n = 40, P_{con} = 0.5, P_{if} = 0.4$ $P_{lp} = 0.6, P_{wait} = 0.3, P_{cre} = 0.3$
(b) $m = 32, P_{con} = 0.5, P_{if} = 0.4$ $P_{lp} = 0.6, P_{wait} = 0.3, P_{cre} = 0.3$
(c) $m = 32, n = 40, P_{con} = 0.5$ $P_{wait} = 0.3, P_{cre} = 0.3$
(d) $m = 32, n = 40, P_{con} = 0.5$ $P_{wait} = 0.3, P_{cre} = 0.3$
(e) $m = 32, n = 40, P_{con} = 0.5$ $P_{if} = 0.4, P_{lp} = 0.6, P_{wait} = 0.3$
(f) $m = 32, n = 40, P_{con} = 0.5$ $P_{if} = 0.4, P_{lp} = 0.6, P_{cre} = 0.3$

Fig. 10: Evaluation results for random task sets.

As shown in Fig. 10(c) and (d), the ratio r of R_1 to R_2 increases when the possibilities p_{lp} and p_{if} increase. The reasons are typically twofold. (1) the upper bound of $vol(\mathcal{T})$ calculated by Alg. 1 becomes more pessimistic when there are more `loop` blocks and more `if-else` blocks that are nested with each other. (2) the upper bound of $len(\mathcal{T})$ calculated by Alg. 3 becomes more pessimistic when there are more `loop` blocks that contain creation vertices. The computation time t_1 and t_2 decrease with the increase of p_{lp}, and increase with the increase of p_{if}. This is because that an `if-else` block may take more time to analyze than a `loop` block since an `if-else` block contains two branches, and comparatively, a `loop` block only has one branch.

As shown in Fig. 10(e), the ratio r of R_1 to R_2 does not change significantly with the increase of p_{cre}. As we know that the larger p_{cre} indicates the less vertices in a task when the creation vertices in the task is fixed. With less vertices, the WCRT bound R_1 and R_2 both decrease, and the trend of r in Fig. 10(e) indicates that R_1 and R_2 descend at the same rate. As shown in Fig. 10(f), the ratio r decreases with the increase of p_{wait}. This is because that the number of wait vertices in a task does not affect the upper bound of $len(\mathcal{T})$ calculated by Alg. 3, but the exact length $len(\mathcal{T})$ becomes larger when there are more wait vertices. The computation time t_1 and t_2 does not change significantly with the increase of p_{cre} and p_{wait}.

B. Realistic OpenMP Programs

We collect 12 OpenMP programs from the BOTS benchmark suite [24], and develop the ompTG tool to transform them into DCG topologies. The framework of ompTG is shown in Fig. 11. First, we analyze the OpenMP program, and slice the code region for each task τ_k. Then we use the "ALFBackend" tool [31] to translate each task τ_k's code region into `task_k.alf`, an intermediate code with ALF format, which contains both the high level control flow information and the low level executable operation information of a program. Based on the translated ALF file `task_k.alf`, we use SWEET tool [32] to build the control flow graph (CFG) of task τ_k. Besides the topology of task graphs, the weight of each vertex, representing its WCET, is also important information when calculating WCRT bounds. The WCET of a vertex heavily depends on the underlying hardware architecture. The current version of ompTG cannot provide precise WCET information of vertices since it lacks the underlying hardware model. We use the static WCET analysis techniques in SWEET [32] to derive the safe WCET for each vertex of τ_k, which uses the SimpleScalar [33] as the underlying processor architecture. The detailed configuration of SimpleScalar simulator can be found in [33], and we omit it here due to the page limitations. Furthermore, we also use the Frama-C [34] and the flow facts analyzer in SWEET tool to analyze loop bound information.

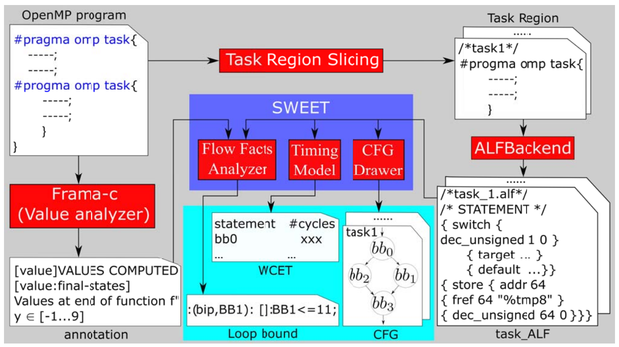


Fig. 11: Framework of ompTG tool.

The first six columns of Table I show the detailed information of the benchmark programs[2]. Columns 2-7 show whether the applications contain a certain structure feature, where T stands for the number of tasks, V stands for the number of vertices, E stands for the number of edges, W stands for the number of `wait` vertices, I and L respectively stand for the number of `if-else` and `loop` structures containing `creation` or `wait` vertices.

The 8-10 columns of Table I give the ratio r of R_1 to R_2 and the computation time t_1 and t_2 for 12 benchmarks with thread number $m = 32$. The bound ratio r for these benchmarks is 3.66 on average. The ratio r for benchmark `sparselu` achieves 30.45, which is the maximum among all benchmarks. This is because that the nesting depth of `loop` blocks is 3 or 4 in `sparselu`, and there are creation vertices in the `loop` blocks with nesting depth 2. Comparatively, In

[2]Some programs have recursive procedures. The problem of dealing with this kind of OpenMP task systems is out of the scope of this paper. In our experiments, we let the iteration numbers of the recursion be 1 for these programs so that they can also fit into our model.

other benchmarks (except `strassen`), the nesting depth of `loop` blocks is 1 or 2, and the creation vertices are only contained in the `loop` block that is not nested in other `loop` blocks. The nesting depth of `loop` blocks in `strassen` is 2 or 3, but there is no creation vertex contained in `loop` blocks. The computation time t_1 and t_2 for these benchmarks are positively correlated with the number of nodes and edges in the task system, which are 14.82 ms and 40.26 ms on average, respectively.

TABLE I: Summary of BOTS programs and evaluation results

program	parameters						$\frac{R_1}{R_2}$	comp time (ms)	
	T	V	E	W	I	L		t_1	t_2
alignment	1	1208	1320	1	45	20	1.00	14.66	40.02
concom	3	126	145	3	3	2	1.81	1.91	4.37
fft	41	8361	8602	18	27	15	1.53	108.73	303.70
fib	8	63	76	2	5	0	1.10	0.77	1.97
floorplan	4	436	473	2	22	8	1.08	3.15	10.48
health	4	412	479	2	21	13	1.07	5.86	14.90
knapsack	7	189	211	2	17	0	1.34	1.94	6.78
nqueens	4	161	180	2	15	9	1.01	1.81	5.00
sort	9	638	725	4	15	3	1.02	10.85	24.77
sparselu	4	544	594	3	5	17	30.45	9.49	19.64
strassen	22	1246	1319	2	9	21	1.00	16.12	44.75
uts	2	191	218	2	7	6	1.49	2.58	6.73

IX. Conclusion

Unlike traditional real-time system models, the workloads generated by OpenMP systems are tightly coupled with program-level structures (e.g., `if-else` and `loops`). Previous work has studied OpenMP tasks with `if-else`, but none of them can model and analyze OpenMP tasks with `loops`. In this paper, we make efforts to expose both `if-else` and `loop` structures in OpenMP tasks. We propose linear-time methods to efficiently compute the two parameters (e.g., the volume and the length) in the WCRT bounds of the OpenMP tasks. Compared with the method that can only compute the parameters roughly, our methods (for computing parameters more exactly) is able to improve the WCRT bound.

Appendix A

Correctness of (17). The paths of $\Pi_{en}^{ct}(B)$ can be represented by the automation $\mathcal{A}_{en}^{ct}$ as shown in Fig. 12.

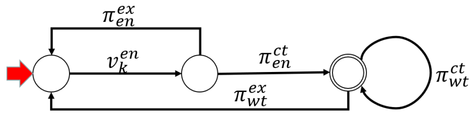


Fig. 12: The automaton $\mathcal{A}_{en}^{ct}$ representing the paths in $\Pi_{en}^{ct}(B)$.

The regular formulation of the path in Π_{en}^{ct} is $v_k^{en}\pi_{en}^{ct}$ $(v_k^{en}\pi_{en}^{ex})^{n_1}(\pi_{wt}^{ct})^{n_2}(\pi_{en}^{ct}\pi_{wt}^{ex}v_k^{en})^{n_3}$. The length $len_{en}^{ct}(B)$ can be calculated as $len_{en}^{ct}(B) = c_k^{en} + len_{en}^{ct}(B') + \Gamma'$, where Γ' is the solution of the following optimization problem.

$$\Gamma' \;\; = \;\; \max\{\alpha n_1 + \beta n_2 + \gamma n_3\} \tag{24}$$

$$s.t. \quad n_1 + n_2 + 2n_3 \leq K - 1 \tag{25}$$

By solving the above problem, we derive (17).

Correctness of (18). We use A_{en}^* in Fig. 13 to show the paths of $\Pi_{en}^*(B)$. The corresponding regular formulation is $(v_k^{en}\pi_{en}^*|v_k^{en}\pi_{en}^{ct}\pi_{wt}^*)(v_k^{en}\pi_{en}^{ex})^{n_1}$ $(\pi_{wt}^{ct})^{n_2}(\pi_{en}^{ct}\pi_{wt}^{ex}v_k^{en})^{n_3}$. The length $len_{en}^*(B)$ is larger than $c_k^{en} + len_{en}^*(B') + \Gamma'$ and $c_k^{en} + len_{en}^{ct}(B') + len_{wt}^*(B') + \Gamma''$, where Γ' is defined by (24) and Γ'' is the solution of the following optimization problem.

$$\Gamma'' \;\; = \;\; \max\{\alpha n_1 + \beta n_2 + \gamma n_3\} \tag{26}$$

$$s.t. \quad n_1 + n_2 + 2n_3 \leq K - 2 \tag{27}$$

By solving the above problem, and since $len_{en}^*(B)$ is no less than len_{en}^{ex} and len_{en}^{ct}, we derive (18).

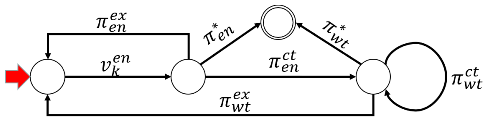


Fig. 13: The automaton $\mathcal{A}_{en}^*$ representing the paths in $\Pi_{en}^*(B)$.

Correctness of (19). The paths of $\Pi_{wt}^{ex}(B)$ are formulated by an automation $\mathcal{A}_{wt}^{ex}$ as shown in Fig. 14, and the corresponding regular formulation of these paths is $\pi_{wt}^{ex}(\pi_{en}^{ex}v_k^{en})^{n_1}(\pi_{wt}^{ct})^{n_2}(\pi_{wt}^{ex}v_k^{en}\pi_{en}^{ct})^{n_3}$. The length $len_{wt}^{ex}(B)$ can be calculated as $len_{wt}^{ex}(B) = len_{wt}^{ex}(B') + \Gamma'$, where Γ' is the solution of the optimization problem in (24) and (25). By solving this problem, we derive (19).

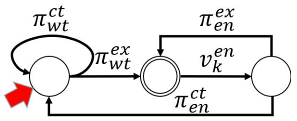


Fig. 14: The automaton $\mathcal{A}_{wt}^{ex}$ representing the paths in $\Pi_{wt}^{ex}(B)$.

Correctness of (20). We formulate the paths of $\Pi_{wt}^{ct}(B)$ as the automation $\mathcal{A}_{wt}^{ct}(B)$ in Fig. 15. The corresponding regular formulation of these paths is $\pi_{wt}^{ct}(\pi_{en}^{ex}v_k^{en})^{n_1}(\pi_{wt}^{ct})^{n_2}(\pi_{wt}^{ex}v_k^{en}\pi_{en}^{ct})^{n_3}$. The length $len_{wt}^{ct}(B) = len_{wt}^{ct}(B') + \Gamma'$, where Γ' is the solution of the optimization problem in (24) and (25). By solving this problem, we derive (20).

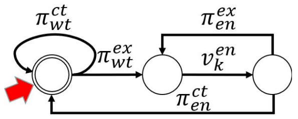


Fig. 15: The automaton $\mathcal{A}_{wt}^{ct}$ representing the paths in $\Pi_{wt}^{ct}(B)$.

Correctness of (21). The paths of $\Pi_{wt}^*(B)$ are formulated as the language of the automation A_{wt}^* in Fig. 16, i.e., $\pi_{wt}^*(\pi_{en}^{ex}v_k^{en})^{n_1}(\pi_{wt}^{ct})^{n_2}(\pi_{wt}^{ex}v_k^{en}\pi_{en}^{ct})^{n_3}$. The length $len_{wt}^*(B) = len_{wt}^*(B') + \Gamma'$, where Γ' is the solution of the optimization problem in (24) and (25). By solving this problem, and since $len_{wt}^*(B) \geq len_{wt}^{ex}(B)$ and $len_{wt}^*(B) \geq len_{wt}^{ct}(B)$, we derive (21).

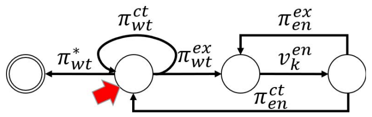


Fig. 16: The automaton $\mathcal{A}_{wt}^*$ representing the paths in $\Pi_{wt}^*(B)$.

Acknowledgment

The first author was supported by the National Foundation of Science of China under Grant 61972076. The corresponding author was supported by Hong Kong Research Grants Council under Grant no. GRF 15213818. The sixth author was supported by the National Foundation of Science of China under Grant U1808206. The authors also thank the anonymous reviewers for their helpful comments.

References

[1] OAR Board. Openmp application program interface version 3.0. In *The OpenMP Forum, Tech. Rep*, 2008.

[2] Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility analysis in the sporadic dag task model. In *Real-time Systems*, 2013.

[3] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435, 2013.

[4] Jing Li, Zheng Luo, David Ferry, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Global edf scheduling for parallel real-time tasks. *Real-Time Systems*, 51(4):395–439, 2015.

[5] David Ferry, Jing Li, Mahesh Mahadevan, Kunal Agrawal, Christopher Gill, and Chenyang Lu. A real-time scheduling service for parallel tasks. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 261–272. IEEE, 2013.

[6] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D Gill. Parallel real-time scheduling of dags. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, 2014.

[7] Sanjoy Baruah. Improved multiprocessor global schedulability analysis of sporadic dag task systems. In *2014 26th Euromicro conference on real-time systems*, pages 97–105. IEEE, 2014.

[8] Manar Qamhieh, Frédéric Fauberteau, Laurent George, and Serge Midonnet. Global edf scheduling of directed acyclic graphs on multiprocessor systems. In *Proceedings of the 21st International conference on Real-Time Networks and Systems*, pages 287–296. ACM, 2013.

[9] Andrea Parri, Alessandro Biondi, and Mauro Marinoni. Response time analysis for g-edf and g-dm scheduling of sporadic dag-tasks with arbitrary deadline. 2015.

[10] Manar Qamhieh, Laurent George, and Serge Midonnet. A stretching algorithm for parallel real-time dag tasks on multiprocessor systems. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, page 13. ACM, 2014.

[11] 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.

[12] Roberto Vargas, Eduardo Quinones, and Andrea Marongiu. Openmp and timing predictability: a possible union? In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 617–620. EDA Consortium, 2015.

[13] Jinghao Sun, Nan Guan, Yang Wang, Qingqiang He, and Wang Yi. Real-time scheduling and analysis of openmp task systems with tied tasks. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 92–103. IEEE, 2017.

[14] Jinghao Sun, Nan Guan, Xu Jiang, Shuangshuang Chang, Zhishan Guo, Qingxu Deng, and Wang Yi. A capacity augmentation bound for real-time constrained-deadline parallel tasks under gedf. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2200–2211, 2018.

[15] Rehan Tariq, Farhan Aadil, Muhammad Faizan Malik, Sadia Ejaz, Muhammad Umair Khan, and Muhammad Fahad Khan. Directed acyclic graph based task scheduling algorithm for heterogeneous systems. In *Proceedings of SAI Intelligent Systems Conference*, pages 936–947. Springer, 2018.

[16] Jian Jia Chen. Federated scheduling admits no constant speedup factors for constrained-deadline dag task systems. *Real-Time Systems*, 52(6):833–838, 2016.

[17] José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Schedulability analysis of dag tasks with arbitrary deadlines under global fixed-priority scheduling. *Real-Time Systems*, 55(1):387–432, 2019.

[18] José Carlos Fonseca, Vincent Nélis, Gurulingesh Raravi, and Luís Miguel Pinho. A multi-dag model for real-time parallel applications with conditional execution. 2015.

[19] Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The global edf scheduling of systems of conditional sporadic dag tasks. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 222–231. IEEE, 2015.

[20] Sanjoy Baruah. The federated scheduling of systems of conditional sporadic dag tasks. In *Proceedings of the 12th International Conference on Embedded Software*, pages 1–10. IEEE Press, 2015.

[21] 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.

[22] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. Schedulability analysis of conditional parallel task graphs in multicore systems. *IEEE Transactions on Computers*, 66(2):339–353, 2016.

[23] Jinghao Sun, Nan Guan, Jingchang Sun, and Yaoyao Chi. Calculating response-time bounds for openmp task systems with conditional branches. In *2019 IEEE 25th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019.

[24] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *2009 international conference on parallel processing*, pages 124–131. IEEE, 2009.

[25] Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[26] Roberto E Vargas, Sara Royuela, Maria A Serrano, Xavi Martorell, and Eduardo Quinones. A lightweight openmp4 run-time for embedded systems. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 43–49. IEEE, 2016.

[27] Jinghao Sun, Nan Guan, Feng Li, Huimin Gao, Chang Shi, and Wang Yi. Real-time scheduling and analysis of openmp dag tasks supporting nested parallelism. *IEEE Transactions on Computers*, 69(9):1335–1348, 2020.

[28] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.

[29] Girija J Narlikar. Scheduling threads for low space requirement and good locality. *Theory of Computing Systems*, 35(2):151–187, 2002.

[30] Tim Paulden and David K Smith. Developing new locality results for the prüfer code using a remarkable linear-time decoding algorithm. *the electronic journal of combinatorics*, pages R55–R55, 2007.

[31] Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg, and Linus Källberg. Alf-a language for wcet flow analysis. In *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[32] Björn Lisper. Sweet–a tool for wcet flow analysis. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 482–485. Springer, 2014.

[33] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, June 1997.

[34] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.