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

Jinhong Sun¹, Nan Guan²†, Zhishan Guo³, Yekai Xue⁴, Jing He¹, Guozhen Tan¹

¹Dalian University of Technology, China; ²City University of Hong Kong, Hong Kong; ³University of Central Florida, U.S; ⁴Northeastern 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 \( \tau_1 \), \( \tau_2 \), and \( \tau_3 \) 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 shown 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.

```c
for(i=0;i<100;i++)
  if(cond-exp)
    #pragma omp task (code1) // \( \tau_1 \)
  else
    #pragma omp task (code2) // \( \tau_2 \)
    #pragma omp task (code3) // \( \tau_3 \)
```

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...
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 \( T \) to denote the set of OpenMP tasks, i.e., \( T = \{ \tau_1, \ldots, \tau_n \} \), where \( \tau_k \) is the \( k \)-th task in \( T \), and \( n = |T| \) is the task number. OpenMP supports nested tasks. More precisely, task \( \tau_1 \) is nested in task \( \tau_k \) if the task directive of \( \tau_1 \) is contained in \( \tau_k \)’s code region. In this case, we say \( \tau_k \) is the parent of \( \tau_1 \), and \( \tau_1 \) is the child of \( \tau_k \). We assume that a task has at most one parent. The task with no parent is called the main task of \( T \). For example, Fig. 2 shows a task set with 4 tasks. The main task \( \tau_1 \) has two children \( \tau_2 \) and \( \tau_4 \), \( \tau_2 \) is the parent of \( \tau_3 \). The code region of \( \tau_1 \) includes Lines 4, 7, 8, 9, 10, 11 and 13.

\(^1\)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.

\[1\] #pragma omp parallel num_threads (m) {
  ...
  #pragma omp single {
      #pragma omp task {/(//creation of \( \tau_1 \))
      #pragma omp task {/(//creation of \( \tau_2 \))
      #pragma omp task {/(//creation of \( \tau_3 \))
      code31: \} \}/v_3^{1}\)
      while (exp1){ //entry: \( v_1^{1}\); exit: \( v_1^{2}\)
          if (exp2){ //entry: \( v_1^{2}\); exit: \( v_1^{3}\)
              #pragma omp taskwait; \}/v_1^{3}\)
              else { \}
                #pragma omp task {/(//creation of \( \tau_4 \))
                code41: \} \}/v_1^{4}\)
                #pragma omp taskwait; \}/v_1^{4}\)
                #pragma omp taskwait; \}/v_1^{5}\)
            #pragma omp taskwait; \}/v_1^{5}\)
          #pragma omp taskwait; \}/v_1^{6}\)
        #pragma omp taskwait; \}/v_1^{7}\)
      #pragma omp taskwait; \}/v_1^{8}\)
    #pragma omp taskwait; \}/v_1^{9}\)
  #pragma omp taskwait; \}/v_1^{10}\)
#}

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 vice versa.

A conditional block is irregular if it contains OpenMP directives (e.g., task and taskwait). We say task system \( T \) is well composed if there is no irregular block in \( T \). Otherwise, \( T \) 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.
Execution Semantics. The execution of OpenMP task system $T$ begins with the execution of its main task $\tau_1$. Other tasks are executed only if its task directive is executed. For any task $\tau_k$ of $T$, the execution of $\tau_k$ means to execute $\tau_k$’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 $\tau_k$ is created, which must be executed later. When encountering a task wait directive, the task $\tau_k$ is suspended until all $\tau_k$’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 which are execution entities (mapped to, e.g., a thread in the whole task system $T$.) BFS and WFS not only lose the work-conserving to execute tasks that have been executed on the threads. The entry vertex of $\tau_i$ is pointed by a CF edge from $\tau_i$’s exit vertex $v_{\tau_i}^e$. The exit vertex of $\tau_i$ points to $\tau_i$’s exit vertex $v_{\tau_i}^e$ via a CF edge. Fig. 2 shows an if-else block with entry vertex $v_1$, exit vertex $v_2$ and two branches: $v_3$ and $v_4$.

• 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 $\tau_i$ with a single entry vertex and a single exit vertex. The entry vertex of $B_i$ is pointed by a CF edge from $\tau_i$’s entry vertex $v_{\tau_i}^e$. The exit vertex of $B_i$ points to $B_i$’s exit vertex $v_{B_i}^e$ via a CF edge. Fig. 2 shows a loop block $B_{lp}$ contains a body $B$, which is a subgraph of $\tau_k$ with a single entry vertex and a single exit vertex. The entry vertex $v_{B_k}^B$ of $B_{lp}$ has two outgoing CE edges: one points to $B$’s entry vertex, and the other one points to $B_{lp}$’s exit vertex $v_{B_k}^{exit}$. $v_{B_k}^B$ and $v_{B_k}^{exit}$ 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 $B_{lp}$ with loop bound $K$ can be unrolled into a directed acyclic (sub)graph as shown in Fig. 3, where the body of $B$ 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.

IV. SYSTEM MODEL

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

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

We distinguish two types of vertices of $\tau_k$: 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 $v_{\tau_k}^{EA}$ and exit vertex $v_{\tau_k}^{EC}$ 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 $\tau_i$ with a single entry vertex and a single exit vertex. The entry vertex of $B_i$ is pointed by a CF edge from $\tau_i$’s entry vertex $v_{\tau_i}^e$. The exit vertex of $B_i$ points to $B_i$’s exit vertex $v_{B_i}^e$ via a CF edge. Fig. 2 shows an if-else block with entry vertex $v_1$, exit vertex $v_2$ and two branches: $v_3$ and $v_4$.

• Loop block $B_{lp}$ contains a body $B$, which is a subgraph of $\tau_k$ with a single entry vertex and a single exit vertex. The entry vertex $v_{B_k}^B$ of $B_{lp}$ has two outgoing CE edges: one points to $B$’s entry vertex, and the other one points to $B_{lp}$’s exit vertex $v_{B_k}^{exit}$. $v_{B_k}^B$ and $v_{B_k}^{exit}$ 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 $B_{lp}$ with loop bound $K$ can be unrolled into a directed acyclic (sub)graph as shown in Fig. 3, where the body of $B$ 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 $v_{\tau_k}^i$ may be contained in several (nested) conditional blocks, we say $v_{\tau_k}^i$ 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 $v_{\tau_k}^i$, which is only closely contained in the if-else block.

Inter-task structure. Recall that the whole OpenMP task system $T$ can be represented by a digraph, which contains the CFG of each task in $T$. The inter-task edges of $T$’s digraph can only connect the task and its child tasks. For the sake of convenience, instead of the whole digraph of $T$, we separately draw the CFG for each task $\tau_k$ of $T$ (with $\tau_k$’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 $v_{\tau_k}^i$ is called the creation vertex if $v_{\tau_k}^i$ represents the task directive that creates a child task $\tau_i$.
of \( \tau_k \). There is a TC edge from the creation vertex \( v_k^1 \) to the source vertex \( v_k^{1|\text{rc}} \) of \( \tau_l \). Fig. 2 shows a creation vertex \( v_l^1 \), which creates \( \tau_2 \). 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 \( v_k^1 \) is called the **wait vertex** if \( v_k^1 \) represents the taskwait directive in the code region of \( \tau_k \). For any task \( \tau_l \) whose creation vertex \( v_l^1 \) is a predecessor of \( v_k^1 \), there is a TW edge from the sink vertex \( v_l^{1|\text{sk}} \) of \( \tau_l \) to the wait vertex \( v_k^1 \). As shown in Fig. 2, \( v_k^1 \) is a wait vertex. There is a TW edge from \( v_k^1 \) to \( v_k^2 \), in the sense that \( \tau_2 \)'s creation vertex \( v_k^1 \) is the predecessor of \( v_k^2 \). Moreover, there is a TW edge from \( v_k^2 \) to \( v_k^4 \) since \( v_k^2 \) is a predecessor of \( v_k^4 \). (The reason is that there is a path from \( v_k^2 \) to \( v_k^1 \) via the back edge \((v_k^2, v_k^1)\) of the loop block.)

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

**Definition 1** (Execution Flow). The execution flow (EF) \( \epsilon \) of \( T \) (written as \( \epsilon \vdash T \)) is a subgraph of \( T \)'s DAG such that the source vertex \( v_{\text{src}}^1 \) of the main task \( \tau_1 \) must belong to \( \epsilon \), and for any vertex \( v_k^j \) that has been involved in \( \epsilon \), if \( v_k^j \) is an entry vertex of a conditional block, only one of its immediate successors is involved in \( \epsilon \). Otherwise, all of its immediate successors (via CF edges and TC edges) are involved in \( \epsilon \). An edge \((v_k^j, v_k^j')\) of \( T \)'s DAG is involved in \( \epsilon \) only if its ending points \( v_k^j \) and \( v_k^j' \) are both involved in \( \epsilon \).

![Fig. 4: The EF of the OpenMP task system T in Fig. 2.](image)

Fig. 4 shows an example EF \( \epsilon \) of \( T \) in Fig. 2, which starts with the source vertex \( v_{\text{src}}^1 \) of \( \tau_1 \). The task \( \tau_2, \tau_3 \) and \( \tau_4 \) (as well as their corresponding TC edges) are involved in \( \epsilon \) since their creation vertices (e.g., \( v_{\text{src}}^1, v_{\text{src}}^2, v_{\text{src}}^3, v_{\text{src}}^4 \)) are involved in \( \epsilon \). The body of the loop block is executed twice in \( \epsilon \), and thus, the vertices in the loop block are duplicated. The TW edges between \( \tau_1 \)'s child task and the wait vertices of \( \tau_1 \) are also added into \( \epsilon \).

The execution of \( T \) is equivalent to travel an EF \( \epsilon \) of \( T \), which starts with the source vertex \( v_{\text{src}}^1 \) and ends with the sink vertex \( v_{\text{src}}^{1|\text{rc}} \) of \( \tau_1 \). A vertex becomes **eligible** to be executed only when its predecessors (in \( \epsilon \)) are all completed. When a vertex is completed, all its successors (in \( \epsilon \)) should be executed. In Fig. 4, the vertex \( v_k^4 \) is eligible if the vertices \( v_k^2, v_k^3 \) and \( v_k^5 \) are completed. According to Sec. III, we schedule the EF \( \epsilon \) on \( m \) threads under work-conserving algorithms, in the sense that a thread always tries to execute vertices of \( \epsilon \), and no threads will be left idle if there are vertices eligible for execution.

**V. RESPONSE TIME BOUNDS**

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

\[
R(T) = \max_{\epsilon \vdash T} R(\epsilon)
\]  

(1)

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

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

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

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 \( \epsilon \) is 5, and the longest path of \( \epsilon \) is \( \pi = (v_1^1, v_1^2, v_1^3, v_1^4, v_1^5, v_1^6) \). The volume of \( \epsilon \) equals 7. The following theorem gives an upper bound of WCRT \( R(T) \).

**Theorem 1.** Under work-conserving algorithms, tasks of \( T \) executed on \( m \) threads have the WCRT bounded by

\[
R(T) \leq \frac{m - 1}{m} \text{len}(T) + \frac{1}{m} \text{vol}(T)
\]  

(2)

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

\[
R(\epsilon) \leq \frac{m - 1}{m} \text{len}(\epsilon) + \frac{\text{vol}(\epsilon)}{m}
\]  

(3)

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

From Thm. 1, we can separately compute the volume \( \text{vol}(T) \) and the length \( \text{len}(T) \) of \( T \). Then we combine these two parameters together by (2), and eventually obtain the upper bound of WCRT \( R(T) \). Computing the parameters for cyclic digraphs 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 \( T \) 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  is \( \pi_1 = (v^1_1, v^1_2, v^1_1, v^1_2, v^1_1, v^1_1, v^1_1, v^1_1, v^1_1, v^1_1) \) and the longest path of  is \( \pi_2 = (v^1_1, v^1_1, v^1_1, v^1_1, v^1_1, v^1_1, v^1_1, v^1_1, v^1_1, v^1_1) \), both of which have the same length 5.

![Fig. 5: The EFs  and  with if-else block taking the same branch in each iteration of the loop block.](image)

• Case 2. The if-else block nested in the loop block takes different branch in each iteration. We obtain the longest path \( \pi^* = (v^1_1, v^1_2, v^1_1, v^1_2, v^1_1, v^1_1, v^1_1, v^1_1, v^1_1, v^1_1) \) of \( T \) with length 6 when the digraph of \( T \) is executed as the EF  that takes the branch \( v^1_1 \) in the first iteration, and takes the branch \( v^1_1 \) in the second iteration as shown in Fig. 6.

![Fig. 6: The EF  with if-else block taking different branch in each iteration of the loop block.](image)

In sum, we know that the length of \( T \) 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 \( T \) to an equivalent DAG. Then we enumerate all EFs on \( T \)'s DAG, and find the one with the maximum parameter. However, it is impractical since the number of EFs of \( T \) is exponential regarding the number of 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 \( vol(T) \) in Sec. VI, and then compute the length \( len(T) \) 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 \( T \)'s hierarchical structure, we propose a more precise method which is a little complicated, but still remains linear-time complexity.

VI. COMPUTING \( vol(T) \)

A. Approximation Method

A trivial bound for the volume \( vol(T) \) is given below.

\[
vol(T) \leq \sum_{\tau_k \in T} tc_{\max}(\tau_k) \times \sum_{v^j_k \in \tau_k} tt_{\max}(v^j_k) \times c^k_j \tag{4}
\]

where \( tc_{\max}(\tau_k) \) is the maximum number of times that the task \( \tau_k \) can be created in an EF \( \epsilon \) of \( T \). \( tt_{\max}(v^j_k) \) is the maximum number of times that the vertex \( v^j_k \) can be traveled in each execution of \( \tau_k \). These parameters \( tc_{\max}(\tau_k) \) and \( tt_{\max}(v^j_k) \) are calculated by Alg. 1. First, we sort the tasks of \( T \) in the order \( \sigma(T) \) such that for any two tasks \( \tau_k \) and \( \tau_l \) before \( \tau_k \), the \( \tau_k \) is before \( \tau_l \), we denote the order in which the vertices of \( \tau_k \) are sorted topologically (by ignoring all back edges). We compute the maximum traversal times of \( \tau_k \)'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 \( \sigma(T) \) from the first to the last do
2. if \( \tau_k \) is the main task of \( T \) then
3. \[ tc_{\max}(\tau_k) := 1; \]
4. else
5. //suppose that \( \tau_k \) is created by vertex \( v^j_l \) of \( \tau_l \)
6. \[ tc_{\max}(\tau_k) := tt_{\max}(v^j_k) \times tc_{\max}(\tau_l); \]
7. for each vertex \( v^j_k \) of \( \gamma(\tau_k) \) from the first to the last do
8. if \( v^j_k \) is the entry vertex of a loop block \( B_{ip} \) then
9. //denote by \( K \) the loop bound of \( B_{ip} \)
10. \[ tt_{\max}(v^j_k) := ((K + 1)) ; \]
11. else
12. \[ tt_{\max}(v^j_k) := 1; \]
13. if \( v^j_k \) is contained in some loop blocks then
14. //suppose that \( v^j_k \) is closely contained in a loop block \( B_{ip} \) with the exit vertex \( v^{ex}_{ip} \) and the loop bound \( K \)
15. \[ tt_{\max}(v^j_k) := tt_{\max}(v^j_k) \times tt_{\max}(v^{ex}_{ip}) \times K; \]

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

**Pessimism.** Formula (4) involves all vertices of \( T \), 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 \( v^1_1 \) is traveled twice in an EF \( \epsilon \) only if the vertex \( v^1_1 \) is not traveled in \( \epsilon \). By (4), the vertices \( v^1_1 \) and \( v^1_1 \) 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 \( T \).

B. More Exact Method

To compute \( \text{vol}(T) \) 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 \( \tau_k \) of \( T \), the syntax tree \( \text{tr}(\tau_k) \) of \( \tau_k \) is denoted as follows.

- The root node of \( \text{tr}(\tau_k) \) represents \( \tau_k \) itself.
- Each leaf node of \( \text{tr}(\tau_k) \) corresponds to a vertex of \( \tau_k \).
- A non-leaf node \( B \) of \( \text{tr}(\tau_k) \) represents a subgraph of \( \tau_k \):
  - If \( B \) represents an \textit{if-else} block \( B_{if} \), it has four child nodes \( v_{in}^k, v_{en}^k, B_1 \) and \( B_2 \), which separately represent the entry vertex, the exit vertex and the two branches of \( B_{if} \).
  - If \( B \) represents a \textit{loop} block \( B_{lp} \), it has three child nodes \( v_{in}^k, v_{en}^k \) 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 the graph and the rest blocks of \( B \), i.e., \( B_2 = B - B_1 \).

\[ \text{Fig. 7: The syntax tree of } \tau_1 \text{ in Fig. 2.} \]

As shown in Fig. 7, the task \( \tau_1 \) 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 \( \text{tr}(\tau_k) \) 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 \( \text{tr}(\tau_k) \) has at least two child nodes, and thus, the non-leaf nodes of \( \text{tr}(\tau_k) \) is less than that of the complete binary tree if they have the same number of leaf nodes. Moreover, since each leaf node of \( \text{tr}(\tau_k) \) corresponds to a vertex of \( \tau_k \), there are at most \( |V_k| \) non-leaf nodes in \( \text{tr}(\tau_k) \). This completes the proof. \( \square \)

Definition 5 (induced digraph). For any node \( B \) of \( \text{tr}(\tau_k) \) that represents a subgraph \( B \) of \( \tau_k \), 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 \( \text{vol}(B) = \max\{\text{vol}(v_k)|v_k \in G(B) \cap \epsilon, \text{for any } \epsilon \in T\} \).

For example, by considering the subgraph \( B_1 \) in Fig. 7, its induced digraph \( G(B_1) \) contains \( B_1 \) itself, the CFG of \( \tau_4 \) that is created by the vertex \( v_5^\tau \) in \( B_1 \), and the TC and TW edges between the vertices of \( B_1 \) and the vertices in \( \tau_4 \). The volume \( \text{vol}(B_1) \) 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 \( \tau_1 \) is the whole digraph of \( T \), and thus, the volume of \( T \) equals that of \( \tau_1 \)'s induced digraph, i.e., \( \text{vol}(T) = \text{vol}(\tau_1) \). Based on this observation, we can compute the volume \( \text{vol}(T) \) by solving \( \text{vol}(\tau_1) \). We propose a dynamic programming algorithm to compute \( \text{vol}(\tau_1) \) as shown in Alg. 2.

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

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

As shown in Line 1 of Alg. 2, by exploiting the reversed order of \( \sigma(T) \), we compute the volume \( \text{vol}(\tau_k) \) of the digraph induced from each task \( \tau_k \in T \) in an iterative way, i.e., for any task \( \tau_k \) and its child task \( \tau_l \), the volume \( \text{vol}(\tau_l) \) computed for \( \tau_k \) can be used for the volume computation of (the induced digraph of) \( \tau_k \). For each task \( \tau_k \), we travel its syntax tree \( \text{tr}(\tau_k) \) from leaves to root (Line 2), and compute the volume \( \text{vol}(B) \) for each node \( B \) in \( \text{tr}(\tau_k) \). If \( B \) is a leaf of \( \text{tr}(\tau_k) \) that represents a vertex \( v_k^\tau \) of \( \tau_k \), we compute \( \text{vol}(B) \) by distinguishing whether \( v_k^\tau \) is a creation vertex or not (see in Lines 3 to 6). Otherwise, \( B \) represents a non-leaf node of \( \text{tr}(\tau_k) \). 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 \( \text{vol}(B) \) is the summation of \( \text{vol}(B_1) \) and \( \text{vol}(B_2) \). If \( B \) corresponds to an \textit{if-else} block with branches \( B_1 \) and \( B_2 \), the branch of \( B \) with the larger volume contributes to \( \text{vol}(B) \) as shown in Line 10. If \( B \) corresponds to a \textit{loop} block \( B_{lp} \) with the body \( B' \) and loop bound \( K \), we know that the entry vertex \( v_k^\tau \) 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 \( \text{vol}(B) \) is calculated by the equation in Line 12. Alg. 2 returns the volume \( \text{vol}(\tau_1) \) of the main task \( \tau_1 \)'s induced digraph as the volume of \( T \) as shown in Line 13.

Complexity. The volume computation for each task \( \tau_k \) 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 \( T \).
VII. COMPUTING $\text{len}(T)$

A. Approximation Method

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

1. for each task $\tau_k$ in $\sigma(T)$ 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 $v_k$ then
         4. $\text{len}^u(B) := c_k^f$;
         5. if $v_k$ is creation vertex that creates $\tau_1$ then
            6. $\text{len}^u(B) := \text{len}^u(B) + \text{len}^u(\tau_1)$;
      7. else if $B$ is the sequence of blocks $(B_1, B_2)$ then
         8. $\text{len}^u(B) := \text{len}^u(B_1) + \text{len}^u(B_2)$;
      9. else if $B$ is an if-else block then
         10. $\text{len}^u(B) := c_k^{e_n} + c_k^{e_k} + \max\{\text{len}(B_1) \text{len}(B_2)\}$;
   11. else if $B$ is a loop block with body $B'$ then
      12. $\text{len}^u(B) := (K + 1) \times c_k^{e_n} + c_k^{e_k} + K \times \text{len}(B')$;
   13. return $\text{len}(T) := \text{len}^u(\tau_1)$;

- If $B$ is a leaf node of $tr(\tau_k)$ (see in Line 3), we assume that $B$ represents a vertex $v_k$ of $\tau_k$, and there are two possibilities. 1) $v_k$ is a creation vertex that creates the child task $\tau_1$ of $\tau_k$. The length of $B$ is bounded by the summation of $c_k^f$ and the upper bound $\text{len}(\tau_1)$ of the length related to $\tau_1$ as shown in Line 6. 2) $v_k$ is not a creation vertex, and in this case, the length of $B$ is simply bounded by $c_k^f$ 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 $\text{len}(B_1)$ and $\text{len}(B_2)$ 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 $\text{len}(B)$ as shown in Line 10. 3) $B$ represents a loop block $B_{loop}$. The upper bound $\text{len}^u(B)$ is calculated in Line 12 as we know that the entry vertex $v_k^{e_n}$ of $B_{loop}$ is traveled at most $K + 1$ times, and the path of $B_{loop}$’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 $\text{len}(T)$ within $O(2N)$, where $N$ is the total number of vertices in all tasks of $T$.

Pessimism. Alg. 3 cannot solve the length $\text{len}(T)$ exactly. We take the computation of $\text{len}^u(B_{loop})$ for the loop block $B_{loop}$ 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 $B_{loop}$ in Fig. 2 is 2. According to Lines 11 and 12, the length bound $\text{len}^u(B_{loop})$ equals $2 \times \text{len}(B_{if})$, where $B_{if}$ is the if-else block inside $B_{loop}$. The longest path of $B_{if}$ is $(v_1^f, v_2^f, v_1^e)$ with the length 2. Therefore, $\text{len}^u(B_{loop}) = 4$. Actually, the longest path of $B_{loop}$ is $(v_2^f, v_3^f, v_5^e, v_4^e, v_6^e, v_7^e)$ with length 3, which is less than the length bound $\text{len}^u(B_{loop})$ computed by Alg. 3. By using the computation result $\text{len}^u(B_{loop})$, we further derive the length bound $\text{len}^u(B_1) = 5$ for the block $B_1$ (as shown in Fig. 7) according to Lines 7 and 8. Moreover, the length bound $\text{len}^u(v_1^e)$ of the digraph $G(v_1^e)$ induced by $v_1^e$ equals 3 according to Lines 5 and 6. The length bound $\text{len}^u(\tau_1)$ for the main task $\tau_1$ equals $\text{len}^u(v_1^e) + \text{len}^u(B_1) = 8$, according to Lines 7 and 8. According to Line 13, the length bound of $T$ returned by Alg. 3 equals 8. As shown in Case 2 of Example 1, the longest path of $T$ 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 $\tau_k$, it is not sufficient to exactly derive the length of $T$ 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 $\text{len}(T)$ 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 $v_k^{e_n}$ and an exit vertex $v_k^{e_k}$ as follows.

- $\Pi_{en}(B)$: the set of paths that start with the entry vertex $v_k^{e_n}$ of $B$. For example, the path $(v_1^f, v_2^f)$ is a path in $\Pi_{en}(B_{if})$ for the if-else block $B_{if}$ in Fig. 2.
- $\Pi_{et}(B)$: the set of paths that start with a wait vertex of $B$. For example, the path $(v_1^e, v_1^e, v_2^e, v_1^e, v_1^e)$ is a path in $\Pi_{et}(B_{loop})$ for the loop block $B_{loop}$ in Fig. 2.
- $\Pi_{ex}(B)$: the set of paths that start with an exit vertex $v_k^{e_k}$ of $B$ and end at an exit vertex $v_k^{e_k}$. For example, the path $(v_1^f, v_1^e, v_1^e)$ is a path in $\Pi_{ex}(B_{if})$ for the if-else block $B_{if}$ in Fig. 2.
- $\Pi_{en}(B)$: the set of paths that start with the entry vertex $v_k^{e_n}$ of $B$ and end at the sink vertex of a child task of $\tau_k$. For example,
the path \((v^*_k, v^1, v^2, v^3, v^*_k, v^4, v^*_k, v^1)\) is a path in \(\Pi^*_w(B_{tp})\) for the loop block \(B_{tp}\) in Fig. 2.

The longest path \(\pi\) of \(T\) must travel the paths of the above path sets if \(\pi\) travels \(B\). We illustrate this by considering two possible cases. (1) If \(\pi\) enters into block \(B\) from the entry of \(B\), \(\pi\) contains the sub-path belonging to one of three path sets \(\Pi_{en}\), where \(y = \{*, ex, ct\}\). (2) Otherwise, \(\pi\) must enter into \(B\) from a wait vertex (via wait edges), and in this case, \(\pi\) contains the sub-path belonging to one of three path sets \(\Pi_{wt}\), where \(y = \{*, ex, ct\}\). For any path set \(\Pi^*_w(B)\) for \(x \in \{en, wt\}\) and \(y = \{*, ex, ct\}\), we use \(len^*_w(B)\) to denote the length of the longest path in \(\Pi^*_w(B)\), i.e., \(len^*_w(B) = \max_{\pi^* \in \Pi^*_w(B)} len(\pi^*)\). For any task \(\tau_k \in T\) and for any node \(B\) of \(tr(\tau_k)\), we use the procedure \texttt{complen}(\(B\)) to compute all lengths \(len^*_w(B)\) of \(G(B)\) where \(x \in \{en, wt\}\) and \(y = \{ex, ct, *\}\).

**Algorithm 4:** Framework for computing \(len(T)\).

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

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

![Fig. 8: The precedence order of computation functions.](image)

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

**Procedure of \texttt{complen}(\(B\))**

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

**Case 1.** \(B\) is a vertex \(v^*_k\) of \(\tau_k\), i.e., \(B = v^*_k\). The lengths \(len^*_w(B)\) (for \(y = \{*, ex, ct\}\)) are calculated as follows.

\[
\begin{align*}
\text{len}^*_{en}(v^*_k) & = c^*_k \quad (5) \\
\text{len}^*_{ct}(v^*_k) & = \begin{cases} 
\ell^*_k + \text{len}^*_{en}(\tau_1) & v^*_k \text{ creates } \tau_1, \\
-\infty & \text{else}.
\end{cases} \\
\text{len}^*_{ex}(v^*_k) & = \begin{cases} 
\ell^*_k + \text{len}^*_{ct}(\tau_1) & v^*_k \text{ creates } \tau_1, \\
\ell^*_k & \text{else}.
\end{cases}
\end{align*}
\]

and the lengths \(\text{len}^*_w(v^*_k)\) (for \(y = \{*, ex, ct\}\)) are calculated as follows.

\[
\begin{align*}
\text{len}^*_{en}(v^*_k) & = \begin{cases} 
\text{len}^*_{en}(v^*_k) & v^*_k \text{ is a wait vertex}, \\
-\infty & \text{else}.
\end{cases}
\end{align*}
\]

**Lemma 2.** For any vertex \(v^*_k\) of \(\tau_k\), the lengths \(\text{len}^*_w(v^*_k)\) is correctly computed by (5) to (8).

**Proof.** Since there is a single path in \(\Pi^*_{en}(v^*_k)\) which only contains vertex \(v^*_k\), the length \(len^*_{en}(v^*_k)\) equals \(c^*_k\), and we derive Formula (5). Formula (6) and (7) separately compute \(\text{len}^*_{en}(v^*_k)\) and \(\text{len}^*_{ct}(v^*_k)\) by distinguishing whether \(v^*_k\) is a creation vertex or not. If this is the case, we assume that \(\tau_1\) is the child task of \(\tau_k\) which is created by \(v^*_k\). The path of \(\Pi^*_{en}(v^*_k)\) begins with \(v^*_k\) and travels a path in \(\Pi^*_{en}(\tau_1)\) related to \(\tau_1\). It indicates \(\text{len}^*_{en}(v^*_k) = c^*_k + \text{len}^*_{en}(\tau_1)\). Similarly, the path of \(\Pi^*_{ct}(v^*_k)\) begins with \(v^*_k\) and travels a path in \(\Pi^*_{ct}(\tau_1)\) which indicates \(\text{len}^*_{ct}(v^*_k) = c^*_k + \text{len}^*_{ct}(\tau_1)\). Otherwise, \(v^*_k\) does not create task. In this case, there is no path belonging to \(\Pi^*_{en}(v^*_k)\), and thus, we define the length \(\text{len}^*_{en}(v^*_k)\) as \(-\infty\). Moreover, the path in \(\Pi^*_{en}(v^*_k)\) only travels \(v^*_k\), and thus, \(\text{len}^*_{en}(v^*_k) = c^*_k\). Formula (8) computes \(\text{len}^*_{ct}(v^*_k)\) by distinguishing whether \(v^*_k\) is a wait vertex or not. If this is the case, the path in \(\Pi^*_{ct}(v^*_k)\) begins with \(v^*_k\), and thus, the length \(\text{len}^*_{ct}(v^*_k)\) equals \(\text{len}^*_{en}(v^*_k)\). Otherwise, there is no path in \(\Pi^*_{en}(v^*_k)\), and thus, we define \(\text{len}^*_{en}(v^*_k)\) 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 \(\text{len}^*_w(B)\) (for \(x = \{en, wt\}\) and \(y = \{*, ex, ct\}\)) as follows.

\[
\begin{align*}
\text{len}^*_w(B) & = \max\{\text{len}^*_w(B_1) + \text{len}^*_w(B_2), \text{len}^*_w(B_1) + \text{len}^*_w(B_2)\} \\
\text{len}^*_w(B_1) + \text{len}^*_w(B_2) \quad (9)
\end{align*}
\]

**Lemma 3.** For any block \(B = (B_1, B_2)\), the lengths \(\text{len}^*_w(B)\) is correctly computed by (9).

**Proof.** The path \(\pi\) in \(\Pi^*_w(B)\) has two possibilities. 1) \(\pi\) only passes one of \(G(B_1)\) and \(G(B_2)\), i.e., \(\pi \in \Pi^*_w(B_1) \cup \Pi^*_w(B_2)\). It indicates that \(\text{len}^*_w(B) \geq \max\{\text{len}^*_w(B_1), \text{len}^*_w(B_2)\}\). 2) \(\pi\) passes both \(G(B_1)\) and \(G(B_2)\), and can be divided into two parts: \(\pi = (\pi_1, \pi_2)\), where \(\pi_1\) is the path which only travels \(G(B_1)\), and \(\pi_2\) is the path which only travels \(G(B_2)\). There are two sub-cases: a) \(\pi_1\) ends at the exit vertex of \(B_1\), i.e., \(\pi_1 \in \Pi^*_w(B_1)\), 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 $\pi_1$ and $\pi_2$, $\pi_2$ must begin with the entry vertex of $B_2$, i.e., $\pi_2 \in \Pi^p_{en}(B_2)$. It indicates that $len^p_{en}(B) \geq len^p_{ct}(B_1) + len^p_{en}(B_2)$. If $\pi_1$ ends at the sink vertex of a child task $\tau$ that is created by a vertex in $B_1$, i.e., $\pi_1 \in \Pi^p_{ct}(B_1)$. Since only wait vertices of $B_2$ can be connected to the sink vertex of $\pi_1$ (via TW edges), to guarantee the connectivity between $\pi_1$ and $\pi_2$, $\pi_2$ must begin with a wait vertex in $B_2$, i.e., $\pi_2 \in \Pi^w_{en}(B_2)$. It indicates that $len^w_{en}(B) \geq len^w_{ct}(B_1) + len^w_{en}(B_2)$. In sum, we can derive (9).

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

\[
\begin{align*}
len^p_{en}(B) &= e^e_k + len^e_{en}(B) + L^e_{en} \quad (10) \\
len^p_{ct}(B) &= e^e_k + L^e_{ct} \quad (11) \\
len^p_{ex}(B) &= \max\{len^e_{en}(B), e^e_k + L^e_{en}\} \quad (12) \\
len^p_{ct}(B) &= L^e_{ct} \quad (13) \\
len^p_{ex}(B) &= \max\{L^e_{ct}, len^e_{ct}(B)\} \quad (15)
\end{align*}
\]

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

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

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

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

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

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

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

\[
\begin{align*}
len^p_{en}(B) &= c^e_k + len^p_{en}(B') + \max\{\Gamma_1, \Gamma_2\} \quad (16) \\
len^p_{ct}(B) &= c^e_k + len^p_{ct}(B') + \max\{\Gamma_3, \Gamma_4\} \quad (17) \\
len^p_{ex}(B) &= \max\{len^p_{en}(B), len^p_{ct}(B), A_1, A_2\} \quad (18) \\
len^p_{ct}(B) &= len^p_{ct}(B') + \max\{\Gamma_3, \Gamma_4\} \quad (19) \\
len^p_{ex}(B) &= len^p_{ct}(B') + \max\{\Gamma_3, \Gamma_4\} \quad (20) \\
len^p_{ex}(B) &= \max\{len^p_{ex}(B), len^p_{ct}(B)\} \quad (21)
\end{align*}
\]

where

\[
\begin{align*}
\alpha &= c^e_k + len^p_{en}(B') \\
\beta &= len^p_{ct}(B') \\
\gamma &= c^e_k + len^p_{ct}(B') + len^p_{ex}(B') \\
A_1 &= c^e_k + len^p_{en}(B') + \max\{\Gamma_5, \Gamma_6\} \\
A_2 &= c^e_k + len^p_{ct}(B') + len^p_{ct}(B') + \max\{\Gamma_5, \Gamma_6\} \\
A_3 &= len^p_{ct}(B') + \max\{\Gamma_3, \Gamma_4\} \\
\Gamma_1 &= \alpha K \\
\Gamma_2 &= (\frac{K}{2} + \frac{K}{2}) \max\{\alpha, \beta\} \\
\Gamma_3 &= \alpha (k-1) \\
\Gamma_4 &= (\frac{K-1}{2} + \frac{K-1}{2}) \max\{\alpha, \beta\} \\
\Gamma_5 &= \alpha (k-2) \\
\Gamma_6 &= (\frac{K-2}{2} + \frac{K-2}{2}) \max\{\alpha, \beta\}
\end{align*}
\]

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 $len^p_{en}(B)$ is correctly computed by (16).

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

Fig. 9: The automaton $A^p_n$ representing the paths in $\Pi^p_{en}(B)$. Each edge of $A^p_n$ is associated with a label such as $v^e_k$, $v^e_k$ and $\pi^e_k$ which denotes the longest path in the path set $\Pi^p_{en}(B')$ 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 $A^p_n$. Each path in $\Pi^p_{en}(B)$ can be represented as a path $A^p_{en}(B)$ from the initial state to the final state. The regular formulation of the paths in $\Pi^p_{en}(B)$ is

\[
v^e_k \pi^e_k v^e_k (v^e_k \pi^e_k)^n (\pi^e_k v^e_k)^2 (\pi^e_k v^e_k)^n\pi^e_k (\pi^e_k v^e_k)^n\pi^e_k (\pi^e_k v^e_k)^n, \text{ which means that}
\]
the vertices $v^i_k$ and $v^v_k$ are separately traveled once; and the segments $\pi_1 = \langle v^v_i, v^e_i \rangle$, $\pi_2 = \langle v^e_i, v^i_k \rangle$, and $\pi_3 = \langle v^i_k, v^e_i \rangle$ 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 $\text{len}_{\text{ex}}(B)$ of the longest path in $\Pi_{\text{ex}}(B)$ can be calculated as $\text{len}_{\text{ex}}(B) = c^e_i + c^v_i + \Gamma$, where $\Gamma$ is the solution of the following optimization problem:

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

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

where the objective function (22) solves the maximum workload of traveling of the segments $\pi_1, \pi_2$ and $\pi_3$. The constraint of (23) ensures the loop bound $K$. Since $\text{len}_{\text{ex}}(B') \geq \text{len}_{\text{ex}}(B)$, 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., $\Gamma = \frac{\gamma K}{2}$. (2) $K$ is odd, the objective function (22) achieves its supremum if $n_2 = \frac{K}{2}$, and $n_1 = 1$ and $n_2 = 0$ if $\alpha > \beta$ or otherwise, $n_2 = 1$ and $n_1 = 0$. We have $\Gamma = \gamma \frac{K}{2} + \frac{n_1 n_2}{2}$, and thus, $\Gamma = \Gamma_2$.

In sum, we derive (16).

**Complexity.** The length computation procedure $\text{complex}(B)$ implemented in this sub-section should compute 6 parameters $\text{len}_{\text{ex}}(B)$ 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 $T$ costs $O(12N)$, where $N$ is the total number of vertices in $T$.

**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., $\rho = \frac{R_1}{R_2}$. 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 $\tau_1$. The leaf node of the tree represents a task that does not have a 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 $\tau_k$ of $\mathcal{T}$, we assume that $\tau_k$ has $n_k$ child tasks, and we randomly generate a CFG of $\tau_k$ with about $K = \frac{n_k}{p_{\text{pre}}}$ non-conditional vertices, where $p_{\text{pre}}$ is the possibility that a non-conditional vertex of $\tau_k$ is a creation vertex. At the very beginning, we generate a single vertex, and add it into the vertex set $V_{\text{new}}$. In each round, for each newly generated vertex $v^v_k$ in $V_{\text{new}}$, we generate the successor vertex of $v^v_k$ with the possibility $p_{\text{succ}} = \max\{0, 1 - \frac{\gamma}{n_{\text{con}}}\}$, where $n_{\text{con}}$ is the number of non-conditional vertices in the current CFG of $\tau_k$. Moreover, we replace the vertex $v^v_k$ by a conditional block with the possibility $p_{\text{cond}}$. Furthermore, we let $p_{\text{cond}} = p_{\text{if}} + p_{\text{else}}$, where $p_{\text{if}}$ is the possibility that a vertex is replaced by an if-else block, and $p_{\text{else}}$ is the possibility that a vertex is replaced by a loop block. The if-else block that is used to replace $v^v_k$ has an entry vertex, an exit vertex and two branches. Each branch contains a single vertex, which is added into the vertex set $V_{\text{new}}$. The loop block that is used to replace $v^v_k$ has an entry vertex, an exit vertex and a body, which contains a single vertex $v^v_k$. We add the newly generated vertex $v^v_k$ into the vertex set $V_{\text{new}}$. 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 $\tau_k$. We randomly select $n_k$ non-conditional vertices to be the creation vertices, and each points to a child task of $\tau_k$. Since the total number of non-conditional vertices is about $\frac{n_k}{p_{\text{pre}}}$, we know that each conditional vertex is a creation vertex with the possibility $p_{\text{pre}}$. For any conditional vertex $v^v_k$, if a predecessor of $v^v_k$ creates the child task of $\tau_k$, we let $v^v_k$ be a wait vertex with the possibility $p_{\text{wait}}$.

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 $\rho$ 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 $\rho$ 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 tolerate the increase of tasks much better than \( R_1 \). The computation time \( t_1 \) and \( t_2 \) both increase linearly with the increase of \( n \).

![Graph showing the ratio of \( R_1 \) to \( R_2 \) against the task number \( n \)]

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 \( p_{tp} \) and \( p_{tf} \) increase. The reasons are typically twofold. (1) the upper bound of \( \text{vol}(T) \) 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 \( \text{len}(T) \) 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 \( p_{tp} \), and increase with the increase of \( p_{tf} \). 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 \( \text{len}(T) \) calculated by Alg. 3, but the exact length \( \text{len}(T) \) 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 \( \tau_k \). Then we use the “ALFBackend” tool [31] to translate each task \( \tau_k \)’s code region into \text{task}_k.\text{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 \text{task}_k.\text{alf}, we use SWEET tool [32] to build the control flow graph (CFG) of task \( \tau_k \). 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 \( \tau_k \), 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.

![Diagram illustrating the framework of ompTG tool.](image)

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 \text{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 \text{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>
<thead>
<tr>
<th>program</th>
<th>$\Gamma_1$</th>
<th>$\Gamma_2$</th>
<th>comp time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignment</td>
<td>128</td>
<td>1320</td>
<td>1</td>
</tr>
<tr>
<td>concom</td>
<td>126</td>
<td>145</td>
<td>3</td>
</tr>
<tr>
<td>fft</td>
<td>8361</td>
<td>8602</td>
<td>18</td>
</tr>
<tr>
<td>fib</td>
<td>63</td>
<td>76</td>
<td>2</td>
</tr>
<tr>
<td>floorplan</td>
<td>436</td>
<td>473</td>
<td>2</td>
</tr>
<tr>
<td>health</td>
<td>412</td>
<td>479</td>
<td>2</td>
</tr>
<tr>
<td>knapsack</td>
<td>189</td>
<td>211</td>
<td>2</td>
</tr>
<tr>
<td>aeques</td>
<td>161</td>
<td>180</td>
<td>2</td>
</tr>
<tr>
<td>sort</td>
<td>638</td>
<td>725</td>
<td>4</td>
</tr>
<tr>
<td>sparselu</td>
<td>544</td>
<td>594</td>
<td>3</td>
</tr>
<tr>
<td>strassen</td>
<td>1246</td>
<td>1319</td>
<td>2</td>
</tr>
<tr>
<td>uts</td>
<td>191</td>
<td>218</td>
<td>2</td>
</tr>
</tbody>
</table>

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 $\Pi^c_1(B)$ can be represented by the automation $A^c_1$ as shown in Fig. 12.

The regular formulation of the path in $\Pi^c_1(B)$ is $v^c_1 v^c_2 v^c_3 v^c_4$. The length $len^c_1(B)$ can be calculated as $len^c_1(B) = len^c(B) + len^c_1(B') + t_1$, where $t_1$ is the solution of the following optimization problem.

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

s.t. $n_1 + n_2 + 2n_3 \leq K - 1$

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

Correctness of (18). We use $A^c_1$ in Fig. 13 to show the paths of $\Pi^*_1(B)$. The corresponding regular formulation is

$$(v^c_1 v^c_2 v^c_3 v^c_4 v^c_5 v^c_6 v^c_7 v^c_8).$$

The length $len^*_1(B)$ is no less than $len^c_1(B)$ and $len^c_1(B)$, we derive (18).

Correctness of (19). The paths of $\Pi^*_u(B)$ are formulated by an automation $A^*_u(B)$ as shown in Fig. 14, and the corresponding regular formulation of these paths is $\pi_{exu}^c(\pi_{exu}^c(\pi_{exu}^c(\pi_{exu}^c(\pi_{exu}^c(\pi_{exu}^c(\pi_{exu}^c(B)))B))))$. The length $len^*_u(B)$ can be calculated as $len^*_u(B) = len^*_u(B') + t_1$, where $t_1$ is the solution of the optimization problem in (24) and (25). By solving this problem, we derive (19).

Correctness of (20). We formulate the paths of $\Pi^c_u(B)$ as the automation $A^c_u(B)$ in Fig. 15. The corresponding regular formulation of these paths is $\pi_{exu}^c(\pi_{exu}^c(\pi_{exu}^c(\pi_{exu}^c(\pi_{exu}^c(\pi_{exu}^c(\pi_{exu}^c(B)))B))))$. The length $len^c_u(B)$ is no less than $len^*_u(B)$ and $len^*_u(B)$, we derive (21).

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery. S. Downloaded on April 28, 2023 at 17:35:36 UTC from IEEE Xplore. Restrictions apply.
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